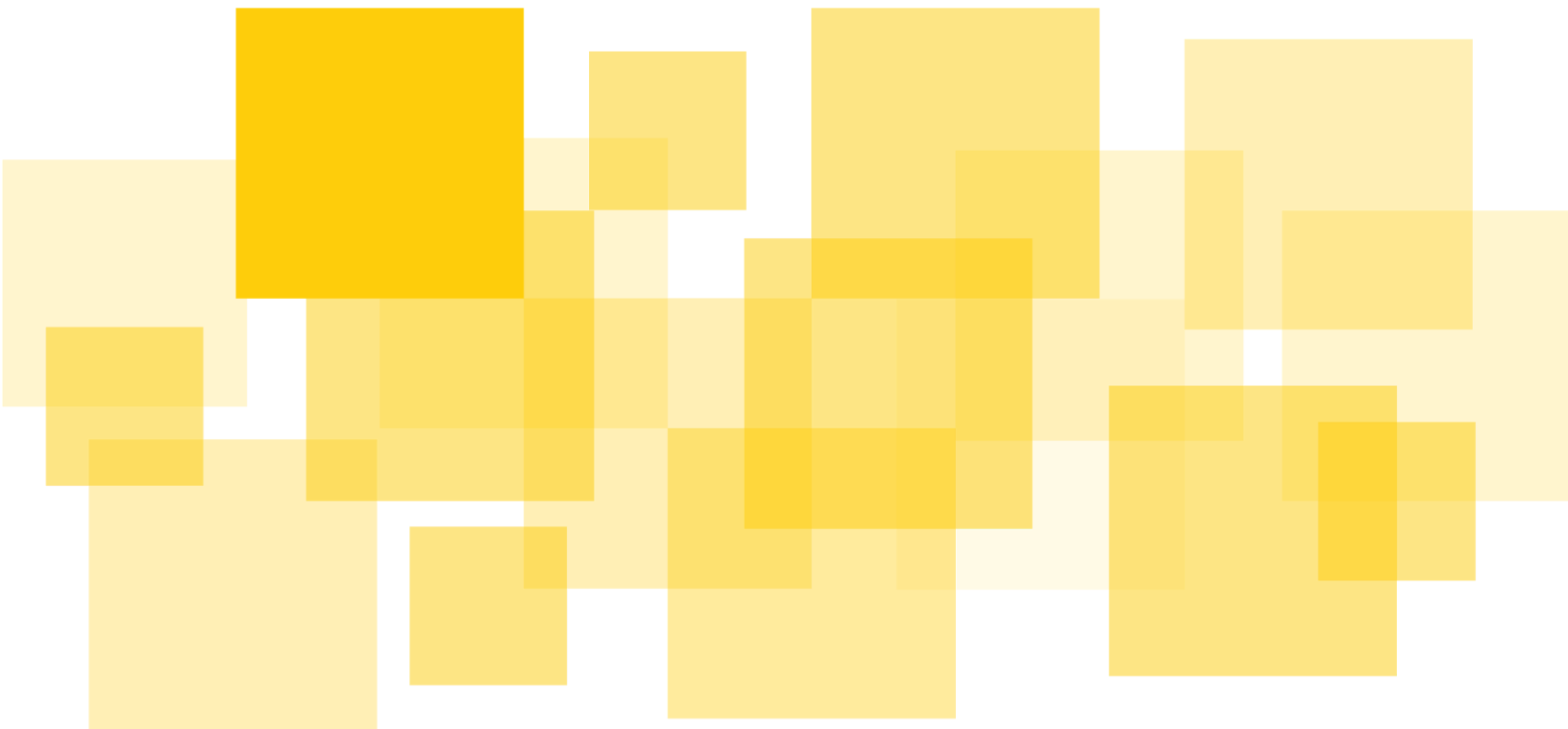


# Security Audit Report

---

## StakerDAO Vault Contract

Delivered: January 30th, 2021



Prepared for StakerDAO by



[Summary](#)

[Disclaimer](#)

[Assumptions](#)

[Findings](#)

[A01: Potential Low-severity DOS Attack Even When User Account Address Unknown](#)

[Informative Findings & Recommendations](#)

[B01: Duplicate GroupSize Check on admin Code Paths](#)

[B02: admin\\_handle\\_update and admin\\_handle\\_delete branches are redundant](#)

[B03: Duplicate txn Accounts 1 == UserVault Check on Non-Admin txn OnCompletion == NoOp Code Paths](#)

[B04: Unnecessary store/load pair On no\\_admin\\_close\\_out Code Path](#)

[B05: Possibly Uninitialized Scratch Space Slot Loaded on no\\_admin\\_burn\\_walogs Code Path](#)

[B06: Unnecessary close\\_out\\_account label](#)

[B07: Typographical Errors in README](#)

[B08: Strategy to Prevent User Algo Loss due to DOS Attack](#)

[B09: Scheme to Prevent Accidental Application Deletion](#)

[Appendix: Local Sandbox Test Setup Script](#)

[Appendix: Table of TEAL Contract Vulnerabilities](#)

[Appendix: Contract Diagram](#)

# Summary

---

[Runtime Verification, Inc.](#) conducted a security audit on the [StakerDAO Vault](#) smart contracts.

The audit was conducted by Stephen Skeirik and Musab Alturki over the course of two calendar weeks. The audit focused on reviewing the business logic of the contracts and identifying any logical loopholes that could cause the system to malfunction or be exploited.

The audit led to one finding and seven informative findings and recommendations. The one finding was a front-running variation of a previously discovered low-severity denial-of-service attack ([A01](#)). The nine informative findings and recommendations are about code size reduction and efficiency ([B01](#), [B02](#), [B03](#), [B04](#)), potential unspecified behavior ([B05](#)), typographical errors ([B06](#), [B07](#)), and additional system safe-guards ([B08](#), [B09](#)).

## Scope


The target of the audit is the smart contracts source files at git-commit-id [040fb0d0e106e9e527eef27e9f615a8077f7126d](#). Below is the list of the source files:

- app-vault-clear-state.teal
- app-vault.teal.tmpl
- minter.teal.tmpl
- vault.teal.tmpl

The audit is limited in scope within the boundary of the TEAL contract code only. Off-chain and client-side portions of the codebase as well as deployment and upgrade scripts are *not* in the scope of this engagement. We did however take some time to review the README file, though we do not consider this file to be formally part of the audit scope.

## Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in [Disclaimer](#), we have followed the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and



the implementation. Second, we thoroughly reviewed the contract source code to detect any unexpected (and possibly exploitable) behaviors. Thirdly, we reviewed the [TEAL guidelines](#) published by Algorand to check for known issues. Finally, given the nascent TEAL development and auditing community, we reviewed this [list of known Ethereum security vulnerabilities and attack vectors](#) and checked whether they apply to TEAL smart contracts; if they apply, we checked whether the code is vulnerable to them. We list our results regarding this analysis in the appendix.



## Disclaimer

---

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

# Assumptions

---

This audit is based on the following assumptions and trust model.

The authorized users (i.e., Admins and Minters) are assumed to behave correctly and honestly, where they are given the following authority:

- Admins may change all logic and parameters in the system, including setting new admins and minters, disabling the application, updating which Alogrand asset (ASA) is used by the system, as well as updating the TEAL code of the application and deleting the application.
- Minters manage the total supply of underlying ASA used by the application.

The deployment scripts are assumed to be correct. Specifically, they are assumed to correctly set up the system parameters including the Admin and Mint account addresses, the underlying ASA ID, and the user vault program prefix and suffix variables. They are also assumed to correctly instantiate all TEAL application parameters including `TMPL_APP_ID`, `TMPL_ASA_ID`, and `TMPL_USER_ADDRESS`.

# Findings

---

Findings presented in this section are issues that can cause the system to fail, malfunction, and/or be exploited, and should be properly addressed.

## A01: Potential Low-Severity DOS Attack Even When User Account Address Unknown

---

Previously, a DOS attack was discovered by which the user could be prevented from opting into the vault application. The documentation was updated to state:

*Attackers can prevent an account to open a Vault if they know the account address on advance. Users should not reveal publicly their accounts that they are going to use to open a Vault.*

This attack occurs as follows:

1. The attacker computes the address `UserVaultAddr` corresponding to the user vault stateless TEAL program;
2. The attacker sends a payment transaction to `UserVaultAddr` with a non-zero amount.

However, a front-running variation of this attack exists even when the attacker does not know the user account address in advance.

We do note that the economic incentives for this kind of attack depend purely on how much money this service earns (which is indirectly bounded by the number of active Algorand accounts), since in that case, a competitor may wish to promote an alternative service by discrediting or disabling this service. As long as the potential service fee earnings remain low, there is no economic incentive for this attack to be carried out.

### Scenario

1. The attacker observes that a vault application instance was created and notes the vault application ID.
2. The attacker then monitors the set of pending, uncommitted transactions on an Algorand node.

3. The attacker observes a pending, uncommitted application call transaction addressed to the vault application ID with OnCompletion type OptIn and notes the address Addr of the first entry in the Accounts field of the transaction.
4. The attacker quickly broadcasts a payment transaction to the address Addr (possibly to different Algorand node(s) on which the OptIn transaction was observed) with the hope that the attack transaction either:
  - a. occurs before the OptIn transaction on a node on which the OptIn transaction was observed; or
  - b. is issued to a node where the OptIn transaction was not observed and that node is selected to be a block proposer before one on which the OptIn transaction was observed.
5. The vault application will then reject the pending OptIn application call transaction (as well as any future OptIn calls), preventing the user from participating in the vault system.

## Recommendation

Document that a front-running variation of the previously known attack exists.

Longer term, it may be possible to rewrite the application such that it is no longer possible to invoke an Application Call transaction with type ClearState, which removes any possibility of a ClearState attack, which in turn, would remove the need for a ClearState attack mitigation which enables a DOS attack. We present below a rough sketch of how we believe this could be done. However, we do not warrant that this approach is correct or carries no other unknown risks.

1. The logic of UserVaultAccount is updated to permit only:
  - a. Application Call transactions with OnCompletion types OptIn, NoOp, and ClearOut;
  - b. Payment transactions.
2. In the application transaction group structure, the initial Application Call transaction would be replaced by two transactions:
  - a. The first transaction would be an Application Call transaction which is almost identical to the one used in the current API, except that the transaction sender is the User Vault Account instead of the User Account.
  - b. The second transaction could actually have any type. The important part is that it is sent from the User Account. It's sole purpose is to indicate that the User Account authorized the transaction group (since this transaction must be signed with the User Account's private key). The Vault



Application logic would be updated to check that the sender of this transaction corresponds to the User Account used to generate the User Vault Account.

3. The opt in process for the User Vault Account would use the transaction group structure from part (2) where the first application call transaction has OnCompletion type OptIn. The Vault Application would verify that the User Vault Account is generated properly from the User Account. At that point, the local storage of the User Vault Account would be updated to contain the same storage variables except that some variable (corresponding to variable  $v$  in the previous storage scheme) would point to the User Account address. This state variable would be used in later application calls to ensure that the User Vault Account and User Account properly correspond.
4. The logic of the other user entry points would remain largely unchanged; the only difference would be that the local storage would be attached to the User Vault Account.
5. The logic of the admin account entry points would remain unchanged except that the logic to enable/disable user accounts would need to become more complex to account for the fact that we disable accounts by User Vault Account address instead of User Account address.

We believe that this approach satisfies two important requirements:

1. Each transaction group requires a signature with the User Account private key to be authorized.
2. It leaves no possibility for the user to induce the User Vault Account to send an Application Call transaction with type ClearState.

## Status

The client acknowledged our finding.

# Informative Findings & Recommendations

---

Findings presented in this section are not known to cause system failures, and do not necessarily represent any flaw in the system. However, they may indicate areas where the code deviates from best practices. Due to discussion about previous issues hitting code size limits, we have included information on potential code size reductions.

## B01: Duplicate GroupSize Check on admin Code Paths

---

All admin code paths require the following condition to hold:

```
global GroupSize == 1
```

This check is performed separately on the three following branches (which contain all successful branches on the admin code path):

1. admin\_handle\_delete
2. admin\_handle\_update
3. admin\_handle\_noop

### Recommendation

To reduce the code size and simplify the application logic, we recommend that the single check be hoisted so that it appears prior to each code path.

### Status

By the time this report was issued, this recommendation had already been adopted.

## B02: admin\_handle\_update and admin\_handle\_delete branches are redundant

---

The two branches admin\_handle\_update and admin\_handle\_delete both contained identical program logic:

```
global GroupSize
int 1
==
bnz success
err
```

### Recommendation

Since these branches only perform a `global GroupSize == 1` check, assuming that check is hoisted above the branch point, these branches can both become a no-op.

### Status

By the time this report was issued, this recommendation had already been adopted.

## B03: Duplicate txn Accounts 1 == UserVault Check on Non-Admin txn OnCompletion == NoOp Code Paths

---

All non-admin code paths which occur after the condition `txn OnCompletion == NoOp` has been checked also require `txn Accounts 1 == UserVault`. This check is performed on the three following branches (which contain all successful branches from this point):

1. `no_admin_mint_algos`
2. `no_admin_withdraw_algos`
3. `no_admin_burn_walgos`

### Recommendation

To reduce the code size and simplify the application logic, we recommend that the single check be hoisted so that it appears prior to each code path.

### Status

By the time this report was issued, this recommendation had already been adopted.

## B04: Unnecessary store/load pair On no\_admin\_close\_out Code Path

---

On lines 536-537, the value 0 is stored to scratch space slot 7. The code continues without branching until line 585, where it is loaded.

We know that the pair of this store and load operation is redundant because:

1. No branches occur between lines 536 and 585;
2. The value at slot 7 is never modified between lines 536 and 585;
3. The value at cell 7 is never used before program termination after line 536.

### Recommendation

Delete the int 0 store 7 on lines 536-537. Replace the load 7 instruction on line 585 with int 0.

### Status

By the time this report was issued, this recommendation had already been adopted.

## B05: Possibly Uninitialized Scratch Space Slot Loaded on no\_admin\_burn\_walgos Code Path

---

When taking either the branch on line 897 or on line 940, the code jumps to line 949 where:

1. The immediately following instruction is load 7 on line 952;
2. No corresponding store 7 instruction appeared earlier on either code path.

At the time of writing this report, the [TEAL opcode semantics](#) did not specify what behavior occurs when an uninitialized scratch space slot is loaded. A preliminary investigation of the TEAL VM codebase (especially [this comment in eval.go](#)) indicated that the default type and value of an uninitialized scratch space slot is an unsigned 64-bit integer with value 0. However, upon further analysis of the same source file above, it appears that loading an uninitialized scratch space slot will produce either:

1. The default value 0 when an unsigned 64-bit integer is the argument type expected at this stack location by the operation which consumes this stack value;
2. The default value "" when a byte string is the argument type expected at this stack location by the operation which consumes this stack value.

The VM code seems to allocate memory in each scratch space slot for *both* an unsigned 64-bit integer and a pointer to a byte string, which indicates that, unless the VM performs some kind of static or dynamic type-checking, it is possible that the same value can be used both as an integer and a byte string. We submitted a GitHub issue to clarify this behavior, but it had not been addressed at the time of report submission.

### Recommendation

To the extent possible, we recommend not relying on unspecified behavior. Thus, we recommend that the mentioned scratch space slot be initialized at some point before line 897 on the no\_admin\_burn\_walgos code path.

### Status

By the time this report was issued, this recommendation had already been adopted.



## B06: Unnecessary close\_out\_account label

---

The label close\_out\_account is never referenced by any branching instruction.

### **Recommendation**

All references to the label close\_out\_account should be deleted.

### **Status**

By the time this report was issued, this recommendation had already been adopted.

## B07: Typographical Errors in README

---

During our evaluation of the code base, we noticed a few typographical errors in the included README file.

1. Admin initializeApp
  - a. Incorrectly lists OnCompletion type as UpdateApplication; on this codepath the required type is actually NoOp.
2. Admin setGlobalStatus
  - a. Identifier arg0 appears twice in the argument list.
3. Admin setAccountStaus
  - a. Identifier arg0 appears twice in the argument list.
4. User closeOut
  - a. Incorrectly states that this code path requires the condition "acc0 must be set to User Vault"; actually, the result of txn Accounts 1, which corresponds to identifier acc0, is never checked on this path.
  - b. Incorrectly states that CloseRemainderTo may be set to any address; the code requires that CloseRemainderTo is not set to the ZeroAddress.
  - c. Documentation on this entry point does not list the requirement that the local variable m must be equal to 0, i.e., that all wrapped Algos minted from this account must be burned for this operation to succeed.
5. User burnwALGOs
  - a. Incorrectly states that Tx1 AssetAmount must be equal to Tx0 arg1; this code path never checks txn ApplicationArgs 1, which corresponds to identifier Tx0 arg1.

### Status

The client fixed all typographical errors in the README.





## B08: Strategy to Prevent User Algo Loss due to DOS Attack

---

Given the fact that the current system usage depends on the user vault contract being empty, it is important that the scripts enforce and the documentation state the following strategy which prevents user Algo loss:

Do not submit a payment transaction to the user vault until after the vault application optIn transaction has been confirmed.

Using this strategy, the user's maximum loss due to the known DOS attacks is limited to being locked out of participating in the vault application.

### **Status**

The client has added this notice to the README.

## B09: Scheme to Prevent Accidental Application Deletion

---

Currently, the system must assume that the Admin account will honestly and correctly invoke all admin entry points. However, there is a potential safeguard that may be added in order to prevent the administrator from accidentally and prematurely deleting the application.

The scenario is simple: if the administrator deletes the application at some point while users are still opted into the system, all of the Algos stored in the User Vaults will become irrevocably locked up.

### **Recommendation**

If additional protection against the administrator accidentally deleting the application is desired, one possibility is to add a new global state variable which stores the total number of opted-in users (which must be updated on each opt-in, close out, and clear state call). The clear state program would have to be updated to include this logic. Then, the delete application code path simply adds a check which prevents the admin from deleting the application when the number of opted in users is non-zero. Of course, this logic breaks in case more than  $(2^{64})-1$  users opt in to use the application, but since addresses are only 32 bits in size, this is currently not possible.

We realize that this recommendation conflicts with a previous recommendation that we made. This recommendation's benefits must be weighed against the additional code complexity and code size limits built into the Algorand stateful TEAL contract system.

Of course, in any case, if the administrator has application update privileges, then they are still able to perform all possible actions by simply updating the application logic to whatever they desire, including the ability to negate all built-in protections. Thus, mitigations like the above can only help prevent honest mistakes.

### **Status**

Client acknowledged our recommendation. They had originally considered this implementation strategy but did not select it due to code size limits.



## Appendix: Local Sandbox Test Setup Script

---

Since the script was not very readable when pasted directly into the document, we instead provide a link to [this GitHub gist](#) which contains the script. Note that this script depends on the current values at the time of report writing of the settings.js file included in the project archive.

## Appendix: Table of TEAL Vulnerabilities

This list of potential TEAL contract security vulnerabilities was based on [this list of known potential Ethereum contract security vulnerabilities](#).

Name	TEAL Vulnerability	Comment
<a href="#">Integer Overflow/Underflow</a>	Partial	TEAL arithmetic functions always fail due to overflow/underflow. While this prevents wrap-around errors, it still could lead to logic errors including contracts that always fail (e.g. if a contract counts how many times it is called, once the counter reaches the maximum size, the contract cannot be called anymore).
<a href="#">Arithmetic Precision Errors</a>	Partial	TEAL arithmetic does not have sub-integer precision; logic errors could occur due to accumulated rounding error from the division operation.
<a href="#">Reentrancy Attacks</a>	None	TEAL contracts cannot programmatically invoke other TEAL contracts.

<a href="#">Access Control: Function Visibility</a>	None	TEAL does not have any concept of public/private entry points.
<a href="#">Access Control: Authentication with tx.origin</a>	None	Since TEAL does not permit programmatic contract calls, there is no distinction between the transaction origin and message sender, making this point moot.
<a href="#">Access Control: Custom Signature Verification</a>	Full	Since TEAL logic can be used to implement custom signature verification schemes, if this logic is incorrect, it may lead to privilege escalation.
<a href="#">Access Control: Unprotected Functions</a>	Full	Any other type of TEAL logic used to restrict access to certain functions may contain errors which could lead to privilege escalation.

<a href="#">Code Injection via delegate call</a>	None	TEAL does not have an equivalent to delegatecall.
<a href="#">Signature Replay Attacks</a>	Full	Custom signature verification logic implemented in TEAL may be vulnerable to signature replay attacks.
<a href="#">Unchecked External Calls</a>	None	TEAL does not allow programmatic calls.
<a href="#">Insufficient Gas Attacks</a>	None	TEAL code is never partially executed due to insufficient funds; instead, all TEAL programs have a fixed maximum cost, and the program does not execute at all if its cost estimate exceeds this value. The system cannot be forced into an inconsistent state by providing insufficient funds.

<a href="#">DOS: Unexpected Revert</a>	None	TEAL does not permit programmatic calls; thus, there is no notion of reverting a call which can be exploited.
<a href="#">DOS: Block Gas Limit</a>	None	As mentioned above, TEAL code execution cannot fail due to insufficient funds.
<a href="#">DOS: External Calls without Gas Stipends</a>	None	TEAL does not permit programmatic calls.
<a href="#">Offline Owner</a>	Full	TEAL code often relies on an administrator account to execute; the code may become inoperable in case the administrator account is offline.

<a href="#">Entropy Illusion</a>	Full	TEAL does not have built-in primitives for secure random generation; custom logic for random number generation may be abused.
<a href="#">Privacy Illusion</a>	Full	Like all blockchain languages, TEAL program state is completely public.
<a href="#">Front-Running</a>	Full	Transactions can be observed before they are committed. The probability of an attack succeeding depends on many factors including but not limited to the transaction pool update policy, the Algorand node network structure, the transaction gossip protocol, the number of nodes controlled by the attacker, and the algorithm for selecting block proposers.
<a href="#">Timestamp Manipulation</a>	Conditionally Full	Depending on how the protocol works, a malicious node may be able to influence the timestamp of a proposed block in order to influence the effect of contract logic.



<a href="#">Unexpected Funds</a>	Full	TEAL logic may fail when an account has non-zero balance, but preventing an account from receiving funds is not possible in Algorand.
<a href="#">External Contract Referencing</a>	Partial	While TEAL contracts may not invoke other contracts, they <i>can</i> : 1. Read the state of other applications if the ApplicationID is known; 2. Require submission in a transaction group that references other TEAL contracts. Especially, mechanism (1) can be abused if the other application permits UpdateApplication calls.
<a href="#">Uninitialized Storage Pointers</a>	Unspecified, Currently None	TEAL store slots that are not uninitialized are, in the current VM implementation, zero-valued. Because this behavior is unspecified, it potentially could change in future versions.
<a href="#">Writes to Arbitrary Storage Locations</a>	None	TEAL does not have storage arrays in this sense.

<a href="#">Incorrect Interface</a>	None	TEAL does not have interfaces to functions.
<a href="#">Arbitrary Jumps with Function Variables</a>	None	TEAL does not have function variables.
<a href="#">Variable Shadowing</a>	None	TEAL does not have variables or scopes to enable shadowing.
<a href="#">Assert Violation</a>	None	TEAL does not have a Solidity assert equivalent.

<a href="#">Dirty Higher Order Bits</a>	None	TEAL does not allow converting byte arrays of arbitrary size to a fixed-width by truncating the higher-order bits. The closest comparable case is the btoi operand in TEAL which converts a byte array to an integer, but it panics if the byte array is too large. While TEAL has a substring function for byte-arrays which could truncate them to a specific size, strings, conceptually, have no concept of higher or lower order bits.
<a href="#">Complex Modifiers</a>	None	TEAL has no equivalent to Solidity modifiers.
<a href="#">Outdated Compiler</a>	None	TEAL has no compiler, and the assembler is so simple that any substantive changes are unlikely.

<a href="#"><u>Use of Deprecated Functions</u></a>	Currently None	TEAL currently has no deprecated functions or operators, but this could change.
<a href="#"><u>Use of Experimental Features</u></a>	Currently None	TEAL currently has no experimental features, but this could change.

# Appendix: Contract Diagram

