Tinyman Governance Protocol

Smart Contract TEAL Review

June 12th, 2024

Prepared for:
Tinyman

Prepared by:
Steve Ferrigno (@nullun)

Table of Contents

Disclaimer

This report is provided for informational purposes only and represents a high-level security assessment based on the information and code provided at the time of the review. While best efforts have been made to identify potential security vulnerabilities, this report does not guarantee the absence of any and all security risks. The responsibility for the implementation and execution of recommended actions lies solely with the project team. The author of this report disclaims any liability for actions taken based on its findings and recommendations.

Intro

Tinyman has brought together various companies and individuals from within the Algorand ecosystem to help review their upcoming governance smart contracts as a way to identify any potential issues or mistakes before they go live. With different approaches being undertaken by different members, they hope to cover all the major areas that could be vulnerable to exploitation.

Approach

During my review of the contracts, I will be focusing on the generated TEAL files. I will be less oriented around the operation of each contract or how they operate together, and more interested in how the low level opcodes are used within themselves. A list of the primary files are shown below and will be reviewed based on commit 9d116ff75f3bfcd54d59fecaa287b5f63c2f4b77.

- `tinyman-governance/contracts/`
  - `proposal_voting/build/`
    - `proposal_voting_approval.teal`
    - `proposal_voting_clear_state.teal`
  - `rewards/build`
    - `rewards_approval.teal`
    - `rewards_clear_state.teal`
  - `staking_voting/build/`
    - `staking_voting_approval.teal`
    - `staking_voting_clear_state.teal`
  - `vault/build/`
    - `vault_approval.teal`
    - `vault_clear_state.teal`

Little attention will be paid to the `*_clear_state.teal` files, as they are all identical and do nothing other than successfully return, regardless of arguments or sender. There are no attempts at “housekeeping” should someone perform a ClearState call.

Contracts

I'll begin by detailing my findings which are common across all four smart contracts. These are the result of using the higher level language Tealish, and compiling their logic out into TEAL files. Tealish is a very good example of a higher level language for Algorand. It provides much better maintainability by a team of developers without them all having to have intricate knowledge of the Algorand Virtual Machine, and offers a much faster ability to understand and learn the codebase for new developers.

All Contracts

Version (#pragma)

All of the TEAL contracts have been generated with `#pragma version 9` at the start of their program. AVMv9 was released in June 2023, reducing round times down to 3.3 seconds and introduced resource sharing between grouped transactions. No new opcodes were included in this release.

Whilst Mainnet is currently on AVMv10, the only additional opcodes are `box_splice` and `box_resize`, various `ec_` maths ops, and the additional fields `AssetCreateMinBalance`, `AssetOptInMinBalance`, and `GenesisHash` for the `global` opcode. It's possible the box related opcodes and new global fields could be used to improve the smart contracts efficiency, however given this TEAL is generated from a higher level language I wouldn't expect this decision to result in any security concerns.

Macros (#define)

At the top of all four contracts a series of macros have been defined using the `#define` keyword. In all cases however, this has been used to easily identify constants which are used throughout the program. This is very helpful from a readability perspective and allows us to quickly identify key values used within the code, as well as being able to understand the later uses of these values without having to remember what they are for. It should be noted that these values will be substituted directly in place of where they're referenced when deployed. So decompiling the contracts would not result in the values appearing at the top.

```
7   // 24 * 60 * 60
8   #define DAY 86400
9   #define BYTES_FALSE "\x00"
10  #define BYTES_TRUE "\x80"
11
12  #define PROPOSAL_STATE_WAITING_FOR_APPROVAL 0
13  #define PROPOSAL_STATE_CANCELLED 1
14  #define PROPOSAL_STATE_PENDING 2
15  #define PROPOSAL_STATE_ACTIVE 3
16  #define PROPOSAL_STATE_DEFEATED 4
17  #define PROPOSAL_STATE_SUCCEEDED 5
18  #define PROPOSAL_STATE_EXECUTED 6
```

Router

The router typically refers to the start of the logical TEAL code and is responsible for routing the execution of the program depending on various parameters of the application call transaction being made. In the case of our Tealish generated TEAL, it's generated a router that expects the very first application argument to contain a string describing the call being made. This is then compared to a list of available paths and uses the `match` opcode to determine which path to take.

```
37  // tl:45: router:
38  pushbytes "create_application"
39  pushbytes "update_application"
40  pushbytes "claim_rewards"
41  pushbytes "increase_budget"
42  pushbytes "create_reward_period"
43  pushbytes "set_reward_amount"
44  pushbytes "get_box"
45  pushbytes "set_manager"
46  pushbytes "set_rewards_manager"
47  pushbytes "init"
48  txna ApplicationArgs 0
49  match route_create_application route_update_application rou
50  err
```

Once a path has been chosen, the `OnCompletion` field is checked, and in almost every scenario it's enforced that only `NoOp` is used when calling the contract. The only exception to this is in the Proposal, Rewards, and Staking contracts which allow an `UpdateApplication` call. However the sender is later checked that they are the authorised address stored in the "MANAGER_KEY"global state. Additionally the route_create_application routes are not checked, but don't have any effect on the outcome (excluding `DeleteApplication` which would result in the program immediately being removed from the chain).

```
73  route_create_reward_period:
74      txn OnCompletion; pushint 0; ==; assert
75      txna ApplicationArgs 1
76      txna ApplicationArgs 2
77      txna ApplicationArgs 3; btoi
78      callsub __func__create_reward_period
79      pushint 1; return
```

Constant Blocks (intcblock)

Throughout all 4 smart contracts `pushint` has been used rather than `int`. This isn't necessarily a bad thing, and does typically help with readability when decompiling a program, however it will result in a larger program size. The only exception is when performing an inner transactions, all contracts use `int 0` and `int 1`. I'm not sure if there was more to this choice, however I believe it's the result of using Tealish rather than a decision on these particular contracts.

```
329     // tl:132: account_power_index_2 = extract3(account_power_indexes, ((i + 1) * 8), 8)
330     load 14                                     // account_power_indexes
331     load 28                                     // i
332     pushint 1
333     +
334     pushint 8
335     *
336     pushint 8
337     extract3
338     store 25                                    // account_power_index_2
```

When TEAL is compiled and deployed on-chain any duplicate int values are typically reduced into a single intcblock at the top of the program, with the opcodes `intc_0` to `intc_3`, and`intc N` used to reference them. This helps reduce the size of the program when the same value is used in multiple places. However by using `pushint` rather than `int`, which these contracts use throughout, you're instructing the compiler to leave them exactly as they are, and will not reduce the size of the program.

Emitting Events (log)

```
290     method "cancel_proposal(address,byte[59])"
291     load 11
292     load 10
293     concat
294     concat
295     log
```

Whilst ARC4 compliance isn't supported by these contracts, ARC28 does appear to be used when it comes to emitting events. The `log` opcode is commonly used as a way to "return" a value back to the user, indicating various outcomes after evaluation. In all instances within these contracts, the `method` opcode has been used to generate an event selector from an event signature, followed by concatenating values. Traditionally the `method` opcode is used for generating return selectors based on a return signature, which includes a response type. Since all instances of the `method` opcode omit a response type it results in a warning from the Algorand TEAL compiler.

```
283:11: invalid ARC-4 ABI method signature for method op: Error parsing return type: cannot convert the string "" to an ABI type
290:11: invalid ARC-4 ABI method signature for method op: Error parsing return type: cannot convert the string "" to an ABI type
458:15: invalid ARC-4 ABI method signature for method op: Error parsing return type: cannot convert the string "" to an ABI type
```

Box Storage

The usage of boxes to retrieve and store structured data is primarily achieved using `box_extract` and `box_replace` with hardcoded offsets calculated by Tealish. This results in a reduced chance of error compared to human-written TEAL where the author may miscalculate an offset or be off by 1. Additionally it seems as though when a box is first retrieved ("opened"), the length of the box is compared against the expected size of the data structure contained within it, seemingly as a form of sanity check.

```
375     // tl:161: assert(proposal.is_approved == BYTES_FALSE)
376     load 11; pushint 80; pushint 1; box_extract// proposal.is_approved
377     pushbytes BYTES_FALSE                                  // "\x00"
378     ==
379     assert
```

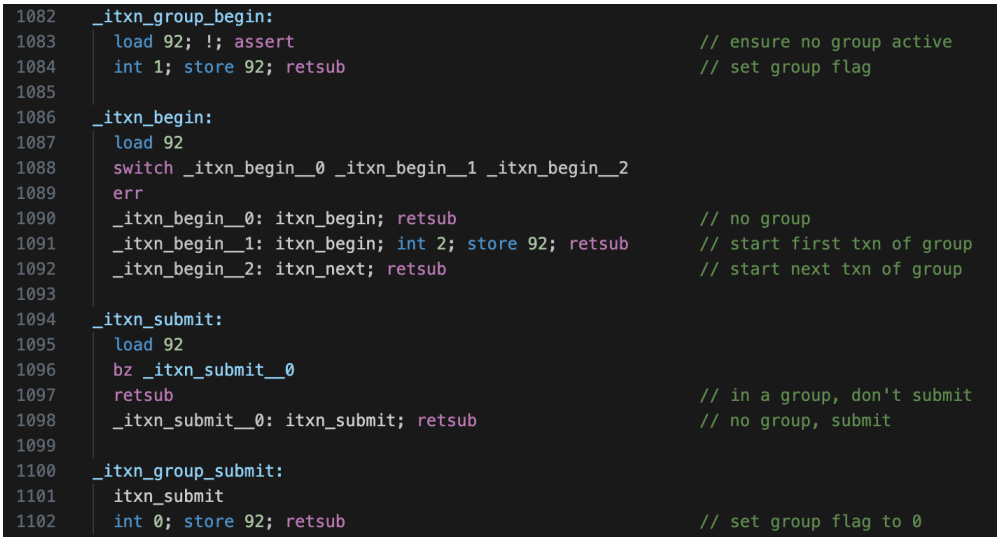
In a limited number of places `box_put` was used to modify the contents of a box. Providing the full data to be written. This either happened shortly after the creation of the box and setting a flag to prevent overwriting existing data, or on data which was not structured within a box.

Increase Opcode Budget

A common pattern for increasing the opcode budget within the AVM is to increase the number of application call transactions submitted within an atomic group. This is typically done by creating a "dummy" transaction which just calls itself and immediately returns true, or by deploying/deleting a brand new app via an inner transaction. An interesting design of these contracts is that only one of the contracts (vault) will actually perform the "dummy" call and return successfully, whilst the other three contracts each have functionality that will call the vault application via inner transactions a number of times as the designated "dummy" transaction destination.

Unused Subroutines

At the bottom of all the smart contracts there appear to be two subroutines which never seem to be called, `_itxn_group_begin` and `_itxn_group_submit`. It's assumed these are helper subroutines introduced by Tealish for making grouped inner transactions, or maybe it was old code that has been left in by mistake. As long as these subroutines were not intended to be used this isn't an issue.



```
1082  _itxn_group_begin:
1083    load 92; !; assert                                  // ensure no group active
1084    int 1; store 92; retsub                             // set group flag
1085
1086  _itxn_begin:
1087    load 92
1088    switch _itxn_begin__0 _itxn_begin__1 _itxn_begin__2
1089    err
1090    _itxn_begin__0: itxn_begin; retsub                  // no group
1091    _itxn_begin__1: itxn_begin; int 2; store 92; retsub // start first txn of group
1092    _itxn_begin__2: itxn_next; retsub                   // start next txn of group
1093
1094  _itxn_submit:
1095    load 92
1096    bz _itxn_submit__0
1097    retsub                                              // in a group, don't submit
1098    _itxn_submit__0: itxn_submit; retsub                // no group, submit
1099
1100  _itxn_group_submit:
1101    itxn_submit
1102    int 0; store 92; retsub                             // set group flag to 0
```

Conclusion

Overall the contracts have been incredibly well designed and the generated TEAL has been presented in a very human friendly format, making it a lot easier to read and understand compared to plain TEAL. Tealish has done a great job at producing human readable TEAL, and while the inclusion of the original comments and code won't be available on-chain -if you were to disassemble the approval program- the structure and layout of the different sections (e.g. router, method functions, common subroutines) are all where you'd expect and doesn't throw you through hundreds of branches with mountains of obfuscated encoding/decoding.

In terms of security, all the contracts follow strict authentication checks where necessary and anytime inner transactions are made prior checks are in place to make sure calls are only being made to their expected destinations. Whether that's sending assets to an account or calling other applications, checks are in place to make sure you're not bypassing intended logic.

Response from Tinyman

After giving the above feedback to Tinyman, they provided responses in the form of verbal explanations or changes to the contracts which will be discussed below. I've summarised the explanations to reduce verbosity.

Constant Blocks (intcblock)

This is intended behaviour, except for the inner transaction subroutines which have since been updated and will be addressed later. So now the compiled TEAL doesn't utilise any `intcblock` opcodes at all, and all ints are pushed explicitly to the stack using `pushint`.

Whilst the original decision may be lost to time, it's most likely that this is a result of needing to have programs that can be disassembled, modified, and reassembled without the algod TEAL compiler attempting to optimise the bytecode and modifying the overall structure of the code. This is important should you have frontends or tooling that generate code from templates.

Emitting Events (log)

Tinyman acknowledges that the warnings -whilst not breaking- can and should be avoided, and have since updated Tealish to include a new ARC28Event feature that replaces the pseudo-opcode `method` directly with `pushbyte` and the 4-bytes return selector of the given return signature. I've verified that all the return selectors generated by this new feature are identical to the original return selectors.

Commit: 35856d3568938bf67c23e69ae20f31b12090d613

Unused Subroutines

The unused subroutines are considered to be "boilerplate" code that's provided by Tealish when compiling a contract. These subroutines have now been removed and have reduced the size of the code.

Commit: e1d63a8581d8390fca7664bf213cbaf4a7d2492b