Tinyman Consensus Staking

Smart Contract TEAL Review

November 21st, 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 consensus staking 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 as well as the on-chain optimised TEAL. 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. A list of the primary files are shown below and will be reviewed based on commit [567dcf508a842f2793b99f8ef655f4da92ab4a88](567dcf508a842f2793b99f8ef655f4da92ab4a88).

- `tinyman-consensus-staking/contracts/`
  - `talgo/build/`
    - `auxapp.teal`
    - `clear_state.teal`
    - `keyreg_logic_sig.teal`
    - `talgo.teal`
  - `talgo_staking/build/`
    - `talgo_staking_approval.teal`
    - `talgo_staking_clear_state.teal`

Little attention will be paid to the `*clear_state.teal` files, as they are 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 the 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 native Algorand Virtual Machine's language TEAL, and offers a much faster ability to understand and learn the codebase for new developers.

All Contracts

Version (#pragma)

All the TEAL contracts have been generated with `#pragma version 10` at the start of their program. AVMv10 was released in January 2024, introducing dynamic round times and ultimately reducing round times below sub-3 seconds, various elliptic curve opcodes were added, as well as `box_splice` and `box_resize`.

For the release of consensus incentives (aka staking rewards) an update to Mainnet will be undertaken and the release of AVMv11 will introduce new opcodes specifically targeting these new features. Firstly `online_stake` will allow you to retrieve the total amount of Algo participating in consensus. Secondly `voter_params_get` provides a way to retrieve details about individual participants status, allowing the retrieval of `VoterBalance` for showing the amount of Algo a particular account is participating with, and `VoterIncentiveEligible` identifying whether the account is eligible for receiving staking rewards, or if they're just participating without rewards. Thirdly new `global` fields have been added:

- `PayoutsEnabled` - Whether block proposal payouts are enabled.
- `PayoutsGoOnlineFee` - The fee required in a keyreg transaction to make an account incentive eligible.
- `PayoutsPercent` - The percentage of transaction fees in a block that can be paid to the block proposer.
- `PayoutsMinBalance` - The minimum algo balance an account must have in the agreement round to receive block payouts in the proposal round.
- `PayoutsMaxBalance` - The maximum algo balance an account can have in the agreement round to receive block payouts in the proposal round.

Finally new `block` fields have been introduced:

- `BlkProposer` - BlkProposer is the Block's proposer.
- `BlkFeesCollected` - BlkFeesCollected is the sum of fees for the block.
- `BlkBonus` - BlkBonus is the extra amount to be paid for the given block (from FeeSink).
- `BlkBranch` - BlkBranch is the hash of the previous block.
- `BlkFeeSink` - BlkFeeSink is the fee sink for the given round.
- `BlkProtocol` - BlkProtocol is the ConsensusVersion of the block.
- `BlkTxnCounter` - BlkTxnCounter is the number of the next transaction after the block.

- `BlkProposerPayout` - BlkProposerPayout is the actual amount moved from feesink to proposer.

Whilst a number of these new opcodes would be beneficial to the contracts, they're certainly not a must-have given the current design, and in order to be deployed to Mainnet prior to the release of the incentives upgrade, the Tinyman team have decided to target AVMv10.

Byte Encoding

Throughout the review process multiple values were found in which escaped strings were used to embed binary data. Whilst I don't believe this poses any direct risk, as any non-ascii characters are escaped appropriately and once deployed on-chain it holds the correct value, it does make reading the generated TEAL slightly more difficult. I recommend these values are instead printed as hexadecimal.

```
// A simple app that rekeys its account to the creator on creation. See auxapp.tl
#define AUX_PROGRAM "\n1\x18\x81\x00\x12D\xb1\x81\x01\xb2\x101\x00\xb2\x071\x00\xb2 \x81\x00\xb2\x01\xb3\x81\x01C"

// This is a LogicSig that allows anything. It is used with change_online_status()
// TEAL: "#pragma version 10\npushint 1\nreturn"
// BYTES: "\n\x81\x01C"
// KEY_REG_LOGIC_SIG_ADDRESS = U3ZXEUNFRSUDPPNFC6U7OBYO4S4AUOEP4RDBI23L2Q5TX3K5LTSVWQOKFM
#define KEY_REG_LOGIC_SIG_ADDRESS "\xa6\xf3rQ\xa5\x8c\xa87\xbd\xa5\x17\xa9\xf7\x07\x0e\xe4\xb8\n8\x8f\xe4F\x14kk\xd4;;\xed]\\\xe5"
```

auxapp.teal

This contract is used exclusively for obtaining additional addresses by the main contract. Note that this contract isn't deployed by a person, but is instead compiled locally and the bytecode embedded within the main contract. It is then deployed by the "parent" smart contract, which during evaluation will rekey the new application address back to the sender ("parent"). Ultimately this gives the parent smart contract an additional address under its control.

OnCompletion

It is recommended that this small smart contract is deployed from talgo.teal using an OnCompletion of `DeleteApplication`, since there is no way (or need for) anyone to interact with the smart contract after its initial use. This helps reduce the minimum balance requirement by the parent application address as it doesn't need to pay for the on-chain storage of the application, as well as reduces the need for every node on the network to store it.

```
itxn_begin

pushint 6 // appl
itxn_field TypeEnum

pushint 0 // NoOp
itxn_field OnCompletion

pushbytes 0x0a311881001244
itxn_field ApprovalProgram

pushbytes 0x0a8101 // 0x0a
itxn_field ClearStateProgra

pushint 1
itxn_field GlobalNumUint

pushint 0
itxn_field Fee

itxn_submit
```

GlobalNumUint

It appears as though an early implementation of this smart contract utilised global state storage, and as a result talgo.teal allocates 1 GlobalNumUint during deployment. This increases the creator minimum balance requirement by 0.0285 Algo. Since this value is never actually set or read by any application, it is

recommended that the parent contract (talgo.teal) deploys the auxapp.teal smart contract without allocating this state.

Optimisation

If the above suggestions are adopted then this smart contract (auxapp.teal) may also be optimised by removing the `txn ApplicationID` value assertion. This certainly isn't necessary, due to all transactions having a fixed fee cost and the contract immediately deleting itself and never actually being stored in the ledger, however in the interest of performance I would be remiss if I didn't point this out.

keyreg_logic_sig.teal

Since the smart contract will be immutable and it's entirely possible that keyreg transactions are changed in the future to include additional fields (as seen with the introduction of state proofs), the smart contract off-loads the keyreg transaction to use a smart signature via the rekey mechanism. This allows the account to submit a keyreg transaction separately from the application, allowing for any additional fields that are currently unknown. The only requirement is that the transaction includes a rekey of the account back to the smart contract.

Misuse of Logic Sig

Due to the talgo.teal smart contract handling all of the transaction verification (e.g. type, sender, rekey, etc) when using the logic sig, what's in the logic sig itself doesn't necessarily matter. However because the current implementation simply uses a `pushint 1; return` program and because this logic sig address is hardcoded into the talgo.teal smart contract, this means anyone can use this logic sig entirely independently of the Consensus Staking platform, and a malicious user could simply rekey this account to themselves to take "ownership" of the account, or they could send transactions to legitimate users with malicious notes/phishing links, appearing to be coming from an "official" Tinyman address. It's worth noting that whilst the logic sig address could be rekeyed by a malicious user, this wouldn't prevent the platform from submitting keyreg transactions, as rekeying to a logic sig address is merely saying "I want my transactions to conform to the TEAL of the logic sig address, and not actually send from the logic sig address".

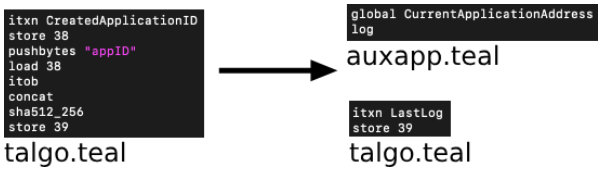
It is my opinion that this approach is more complicated and confusing for end users, and a simpler approach would be to use a more strict logic sig preventing everything from using it, or removing the logic sig all together and using an account already known by the system, for example the node manager who is issuing the application call transaction.

talgo.teal

This smart contract is the main part of the incentives platform. It's responsible for creating the tAlgo asset, setting up the auxiliary accounts, configuring node managers, accepting deposits/withdrawals of Algo, administering the keyreg transactions, as well as numerous other administrative tasks.

Auxiliary Addresses

Upon creating additional accounts the smart contract will calculate the new address by taking the new Application ID, adding "appID" as a prefix, then hashing the result using `sha512_256`. Whilst this is the technically correct way to calculate an address, doing so in TEAL is somewhat costly, coming in at 49 opcode cost. A cheaper solution would be to have the auxapp "return" their address using `log` and then reading the value using `itxn LastLog`, reducing the cost to approximately 3. Similar to the optimisation section for auxapp.teal, this doesn't exactly have any negative impact on the contract, but I'm pointing it out for informational purposes as it's a relatively large reduction.



talgo_staking_approval.teal

The tAlgo staking smart contract allows users to further stake their tAlgo to receiver TINY token. This process swaps the tAlgo asset for an stAlgo asset which cannot be traded (as it's frozen), and a clawback transaction is used when decreasing the position, returning stAlgo back in return for tAlgo. TINY is rewarded to these users over the duration of the staking period, based on their TINY Power provided by another Tinyman platform. The interaction with that platform is out of scope of this report.

Increase Stake with AssetCloseTo

Typically a UI or SDK will handle the construction of the transaction groups being submitted to the smart contract, however if someone were to incorrectly create a transaction which used the `AssetCloseTo` when increasing their stake, any additional tAlgo included outside of the `AssetAmount` value would be lost and not provided as stAlgo. Whilst there is a sanity check with the application argument being the total amount being sent, which is compared to the AssetAmount being received, nothing is preventing a successful transaction with the `AssetCloseTo` being set. You could argue that someone may intentionally set the `AssetCloseTo` if they know their balance will be 0 after the transfer and they want to reduce their minimum balance requirement, however I think the chances of that are slim, and preventing it all together by checking `gtxns AssetCloseTo` is set to `global ZeroAddress` would be safer.

Conclusion

During this review process various points of interest have been identified and reported back to the Tinyman team. Whilst the timeline was tight, the feedback was well received with discussions by all parties on how to improve and mitigate various issues.

I am particularly happy to see yet another immutable design by Tinyman, further pushing the ecosystem towards a more decentralised and permissionless future. Although certain parts of this implementation are currently called by select individuals, there's no reason why this couldn't be delegated to a TINY Governance vote, with additional smart contracts being used for moving the stake and setting node managers.

Response from Tinyman

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

All Contracts

I am pleased to share that the escaped strings are now being represented as hex values. :)

auxapp.teal

All of the suggestions have been taken into consideration and the changes are now reflected in the latest version of the code.

Commits: fddc7a01b658f712f7a4a54300ff6ac09f9f0fc9
ad713206424c759677669b980cc9c2f1be02920d

keyreg_logic_sig.teal

Tinyman has decided to remove the logic sig and instead will have the relevant node manager account act on behalf of the participating account when submitting a keyreg transaction to the network. This means the manager will have the account rekeyed to their account temporarily and within the very next transaction require the keyreg transaction to be submitted whilst also enforcing the rekey of the account back to the talgo.teal smart contract address.

Commit: 3ac56b1913b528feb9021ec076c5d8116ad1ee94