Security Audit Report

Algorand Governance Rewards

Delivered: September 30, 2021
Minor revision: October 14, 2021

Prepared for the Algorand Foundation by

runtime
verification

Contents

Summary

The Algorand Foundation engaged Runtime Verification Inc to conduct a security audit of the smart contracts implementing the Algorand Governance Rewards program.

The objective was to review the contracts' business logic and implementation in PyTeal and identify any issues that could potentially cause the system to malfunction or be exploited.

Timeline

The audit has been conducted under a tight time constraint over a period of 8 working days.

Scope

The audit was conducted by Georgy Lukyanov on the following artefacts provided by the Foundation:

- Rewards Application contracts/rewards_application.py maintains a list of governors and tracks their reward claims;
- Stateless Governance Escrow contracts/logicsig.py is an escrow account that holds the rewards and verifies payment transactions to the eligible governors.

Note that the audit has been performed on the code from a private GitHub repository. The links above refer to a curtailed public repository, but the relevant on-chain contract code only differs in literal constants and one additional assertion to address the finding A01. The additional short logic signature contracts/call_app.teal was not present in the scrutinised private repository and thus is out of scope of the audit.

Findings

Several potential attacks scenarios on the contracts were considered:

- A01. Obtaining control over the escrow
- A02. Multiple claims by the same governor
- A03. Malicious transaction draining escrow through transaction fees

All attack scenarios are either blocked by the validation logic, or the validation logic will be enhanced before deployment.

Additionally, we report several informative findings regarding the contracts design and implementation:

- B01. Delayed reward claim may be compromised
- B02. Unreachable code in compiled `app_approval.teal`

None of the informative findings constitute a threat to the rewards distribution process, but are still worth bringing to the attention of the Foundation and the wider Algorand community.

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 contracts only, and make no material claims or guarantees concerning the contracts' operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of these contracts.

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.

Goals

The goals of the audit are:

- Review the architecture of the governance rewards smart contracts based on the provided documentation;
- Review the PyTeal implementation of the contracts and the compiled TEAL code to identify any programming errors;
- Cross check the compiled TEAL code of the contracts with the documented high-level design.

The audit focuses on trying to identify issues in the system's logic and its implementation that could potentially render the system vulnerable to attacks or cause it to malfunction.

Scope

We audit two smart contracts in the private Algorand Governance Rewards repository as of commit 131c89304ea276c923ad025ec590ff4e1f551c3c. The linked repository remains private. For public scrutiny, the Algorand Foundation has released the contracts' code, and the following links correspond to the public versions of the audited contracts:

- Rewards Application contracts/rewards_application.py maintains a list of governors and tracks their reward claims;
- Stateless Governance Escrow contracts/logicsig.py is an escrow account that holds the rewards and verifies payment transactions to the eligible governors.

The only changes in the publicly released contracts are revised address constants and an additional assertion to address the finding A01.

Additionally, we use the design document for governance rewards smart contracts as a reference.

Methodology

The timeline was very tight, therefore we have only performed a best-effort audit.

Both smart contracts are implemented in PyTeal, a Python EDSL for writing TEAL programs. In order to exclude the PyTeal compiler form the trusted base, we compiled the contracts to TEAL and performed the audit on the compiled code.

We derived a control-flow graph for the contracts using a tool called Tealer. The graphs can be found in appendices to this document.

Basing on the CFG, we built a best-effort model of the contracts' semantics as a transition system embedded in the K Framework. The purpose of modelling was not to build a complete model, but rather to improve our understanding of the contracts' behaviour through the modelling process.

The model abstracts away the low-level details of the contracts, in particular the storage layout which is implemented as bit slices. We model storage as a traditional key-value data structure.

A combination of modelling and manual code review has enabled us to construct the attack scenarios presented above.

To encode the potential attack scenarios, we modified the provided test suite to include the malicious transaction groups. We used a local devnet setup provided by Algorand Foundation to execute the scenarios. We analyse the individual scenarios in further sections.

Attack scenario analysis

After reviewing the design and implementation of the contracts, we identified several possible attack scenarios, targeting two objectives:

- Stealing the funds from the rewards escrow;
- Disrupting the operation of the governance contracts;
- Partially burning the funds of the escrow via fees of the malicious transactions.

All these attacks have either proved to be impossible or the necessary mitigation measures were introduced.

However, the programming pattern we came across in the contracts' implementation has alerted us to perform additional checks, since it was divergent from the official Algorand developer guidelines. In particular, neither of the two contracts checks the size of an incoming transaction group. Therefore, we checked if a number of malicious transactions grouped with the valid ones could be approved. The rest of the section describes these malicious groups in more detail and confirms their denial by the contracts.

A01. Obtaining control over the escrow

The governance escrow will hold the rewards and issue payment to the governors. The Foundation plans to make this account non-participating to exclude it from the consensus and block it from earning rewards.

However, the validation code for this `KeyRegTxn` transaction does not check the `RekeyTo` field, enabling anyone to add an arbitrary authorised address for the escrow. This authorised address, if overlooked, may be used to steal the rewards after the escrow was funded by the rewards pool.

The attack scenario has been brought to our attention by Shai Halevi — a member of the Algorand Foundation. This episode proves the vitality of close contact between the auditors and developers.

Recommendation

Check that the `RekeyTo` field of the transaction to make the escrow non-participating is set to `ZeroAddress`, thus blocking rekying completely. As per the design document, the escrow changes every governance period. Therefore there is no need to support rekeying it, since there is also no need to keep its public address static.

Status

A check that the `KeyRegTxn` transaction does not perform rekeying has been introduced.

A02. Multiple claims by the same governor

A governor could attempt to submit a valid claim and an additional payment transaction, causing them to be payed twice.

We tried executing the attack on the sandboxed devnet provided by the Algorand Foundation. To our surprise, to make the transactions be even considered by the ledger, we had to make them unique, i.e. include random notes in the duplicate malicious transactions.

We tried submitting malicious groups of transactions of size 3 and 4:

Group 1. A valid group with an additional payment transaction: `[pay, appl, pay]`

Group 1 is rejected because there is no accompanying `ApplicationCallTxn` transaction for the extra pay transaction submitted. More specifically, the evaluation of the escrow's TEAL program failed on the additional payment transaction because an attempted out-of-bounds access by the `gtxns` opcode.

Group 2. A valid group with both transactions duplicated: `[pay, appl, pay, appl]`

Group 2 is rejected because by the TEAL approval program of the stateful smart contract. The effects of the first, valid, `ApplicationCallTxn` transaction are being applied to the tentative block; thus the governor's bit, tracking if the reward had already been payed, is already set to one, hence the rejection of the second `ApplicationCallTxn`.

Recommendation

We recommend introducing a `global GroupSize == 2` check into the escrow TEAL program. This would greatly simplify understanding of which transaction groups are considered valid, thus making it easier to ensure security.

Additionally, we suggest adding a negative integration test case describing the attack scenarios presented above.

A03. Malicious transaction draining escrow through transaction fees

Since the group size is not checked, another potential attack scenario was to submit a number of dummy `AssetCreation` transactions, thus forcing the escrow to pay network fees.

The scenario fails for the same reason that A02 does: the alignment of with an `ApplicationCallTxn` is not satisfied.

Recommendation

Same as for A02, we recommend adding a group size check to simplify the make the transaction validation logic easier to understand.

Additional Findings

B01. Delayed reward claim may be compromised

Description

The Algorand Governance FAQ, answering the question Q59, states that governors can claim their rewards at a later time since they may want to delay the claim for tax reasons. However, any third party account can trigger a claim for any eligible governor without their permission, causing the governor to get custody of their reward; thus potentially requiring them to report it to the tax authorities, depending on the jurisdiction.

Status

The Algorand Foundation is aware of this discrepancy. The incentive for a third party to trigger somebody else's claim is deemed to be negligible, since the said party does not gain anything. Allowing an external account being able to trigger an arbitrary governor's claim enables improving the user experience by providing a user-friendly web interface or a similar facility to claim rewards.

B02. Unreachable code in compiled `app_approval.teal`

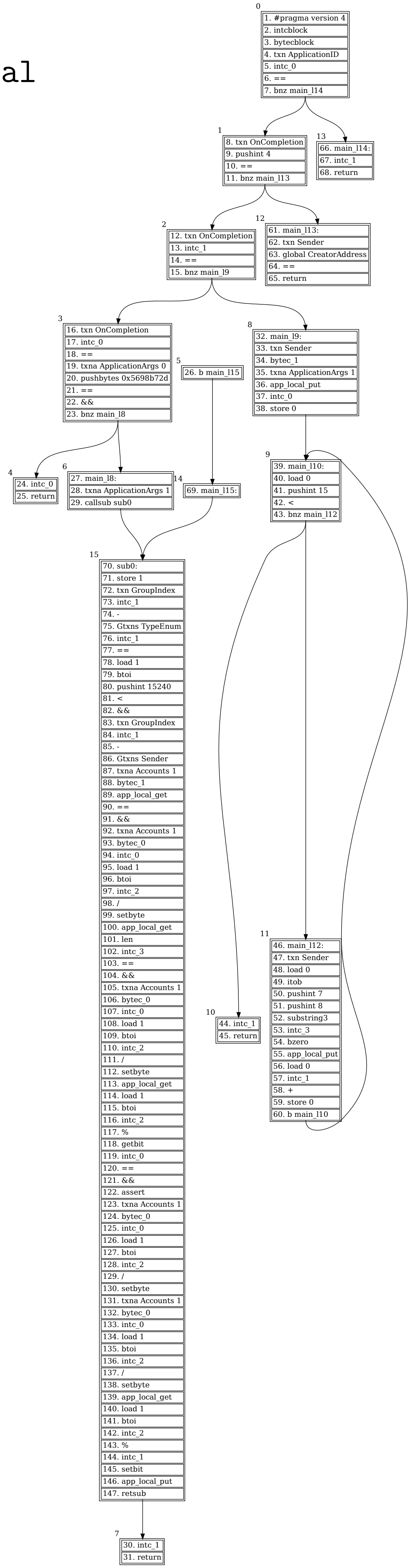
The compiled stateful contract contains an unreachable label as a second entry point to the "claim" subprogram. The label is an artefact of the PyTeal compiler and is harmless; hence this finding is purely informational.

Potentially, dead code could become a problem if the contract size approaches the limits prescribed by AVM.

Appendices

We include the control-flow graphs of the two contracts when compiled to TEAL. Please zoom the page in your PDF viewer to enlarge the graphs. You may want to omit the appendices when printing the report on paper.

app_approval.teal



```
1. #pragma version 4
2. intcblock
3. bytecblock
4. txn ApplicationID
5. intc_0
6. ==
7. bnz main_l14
```

```
8. txn OnCompletion
9. pushint 4
10. ==
11. bnz main_l13
```

```
12. txn OnCompletion
13. intc_1
14. ==
15. bnz main_l9
```

```
16. txn OnCompletion
17. intc_0
18. ==
19. txna ApplicationArgs 0
20. pushbytes 0x5698b72d
21. ==
22. &&
23. bnz main_l8
```

```
24. intc_0
25. return
```

```
26. b main_l15
```

```
27. main_l8:
28. txna ApplicationArgs 1
29. callsub sub0
```

```
30. intc_1
31. return
```

```
32. main_l9:
33. txn Sender
34. bytec_1
35. txna ApplicationArgs 1
36. app_local_put
37. intc_0
38. store 0
```

```
39. main_l10:
40. load 0
41. pushint 15
42. <
43. bnz main_l12
```

```
44. intc_1
45. return
```

```
46. main_l12:
47. txn Sender
48. load 0
49. itob
50. pushint 7
51. pushint 8
52. substring3
53. intc_3
54. bzero
55. app_local_put
56. load 0
57. intc_1
58. +
59. store 0
60. b main_l10
```

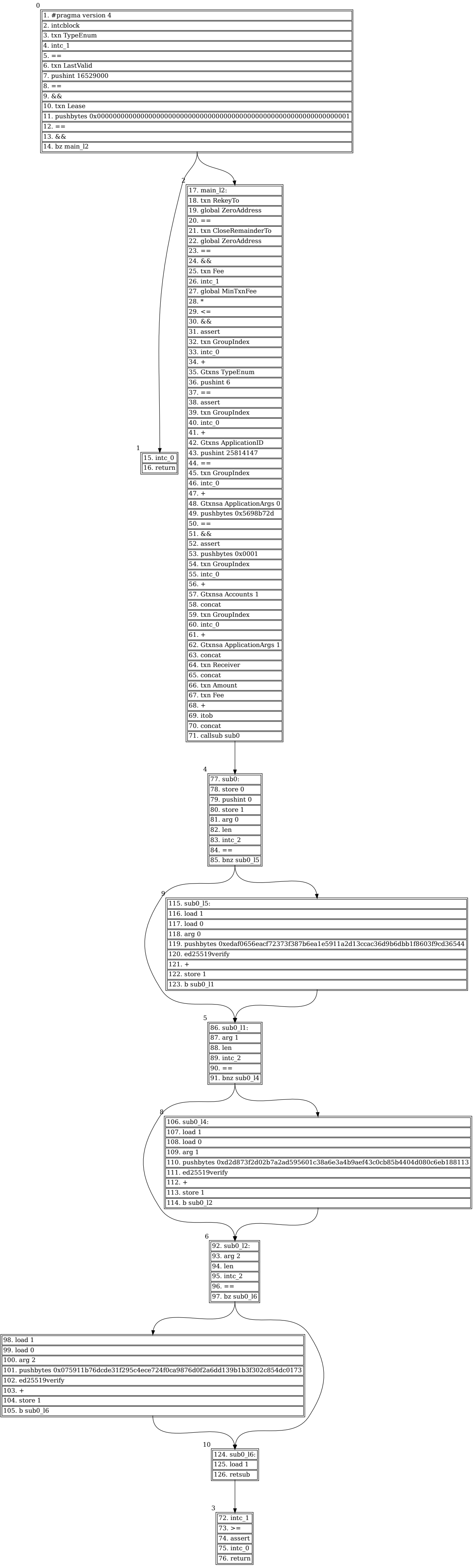
```
61. main_l13:
62. txn Sender
63. global CreatorAddress
64. ==
65. return
```

```
66. main_l14:
67. intc_1
68. return
```

```
69. main_l15:
```

```
70. sub0:
71. store 1
72. txn GroupIndex
73. intc_1
74. -
75. Gtxns TypeEnum
76. intc_1
77. ==
78. load 1
79. btoi
80. pushint 15240
81. <
82. &&
83. txn GroupIndex
84. intc_1
85. -
86. Gtxns Sender
87. txna Accounts 1
88. bytec_1
89. app_local_get
90. ==
91. &&
92. txna Accounts 1
93. bytec_0
94. intc_0
95. load 1
96. btoi
97. intc_2
98. /
99. setbyte
100. app_local_get
101. len
102. intc_3
103. ==
104. &&
105. txna Accounts 1
106. bytec_0
107. intc_0
108. load 1
109. btoi
110. intc_2
111. /
112. setbyte
113. app_local_get
114. load 1
115. btoi
116. intc_2
117. %
118. getbit
119. intc_0
120. ==
121. &&
122. assert
123. txna Accounts 1
124. bytec_0
125. intc_0
126. load 1
127. btoi
128. intc_2
129. /
130. setbyte
131. txna Accounts 1
132. bytec_0
133. intc_0
134. load 1
135. btoi
136. intc_2
137. /
138. setbyte
139. app_local_get
140. load 1
141. btoi
142. intc_2
143. %
144. intc_1
145. setbit
146. app_local_put
147. retsub
```

logicsig.teal



```
1. #pragma version 4
2. intcblock
3. txn TypeEnum
4. intc_1
5. ==
6. txn LastValid
7. pushint 16529000
8. ==
9. &&
10. txn Lease
11. pushbytes 0x0000000000000000000000000000000000000000000000000000000000000001
12. ==
13. &&
14. bz main_l2
```

```
17. main_l2:
18. txn RekeyTo
19. global ZeroAddress
20. ==
21. txn CloseRemainderTo
22. global ZeroAddress
23. ==
24. &&
25. txn Fee
26. intc_1
27. global MinTxnFee
28. *
29. <=
30. &&
31. assert
32. txn GroupIndex
33. intc_0
34. +
35. Gtxns TypeEnum
36. pushint 6
37. ==
38. assert
39. txn GroupIndex
40. intc_0
41. +
42. Gtxns ApplicationID
43. pushint 25814147
44. ==
45. txn GroupIndex
46. intc_0
47. +
48. Gtxnsa ApplicationArgs 0
49. pushbytes 0x5698b72d
50. ==
51. &&
52. assert
53. pushbytes 0x0001
54. txn GroupIndex
55. intc_0
56. +
57. Gtxnsa Accounts 1
58. concat
59. txn GroupIndex
60. intc_0
61. +
62. Gtxnsa ApplicationArgs 1
63. concat
64. txn Receiver
65. concat
66. txn Amount
67. txn Fee
68. +
69. itob
70. concat
71. callsub sub0
```

```
15. intc_0
16. return
```

```
77. sub0:
78. store 0
79. pushint 0
80. store 1
81. arg 0
82. len
83. intc_2
84. ==
85. bnz sub0_l5
```

```
115. sub0_l5:
116. load 1
117. load 0
118. arg 0
119. pushbytes 0xedaf0656eacf72373f387b6ea1e5911a2d13ccac36d9b6dbb1f8603f9cd36544
120. ed25519verify
121. +
122. store 1
123. b sub0_l1
```

```
86. sub0_l1:
87. arg 1
88. len
89. intc_2
90. ==
91. bnz sub0_l4
```

```
106. sub0_l4:
107. load 1
108. load 0
109. arg 1
110. pushbytes 0xd2d873f2d02b7a2ad595601c38a6e3a4b9aef43c0cb85b4404d080c6eb188113
111. ed25519verify
112. +
113. store 1
114. b sub0_l2
```

```
92. sub0_l2:
93. arg 2
94. len
95. intc_2
96. ==
97. bz sub0_l6
```

```
98. load 1
99. load 0
100. arg 2
101. pushbytes 0x075911b76dcde31f295c4ece724f0ca9876d0f2a6dd139b1b3f302c854dc0173
102. ed25519verify
103. +
104. store 1
105. b sub0_l6
```

```
124. sub0_l6:
125. load 1
126. retsub
```

```
72. intc_1
73. >=
74. assert
75. intc_0
76. return
```