



Yieldly.Finance Bridge Algorand

Smart Contracts Security Audit

Prepared by: Halborn

Date of Engagement: May 19th-31st, 2021

Visit: Halborn.com

DOCUMENT REVISION HISTORY	2
CONTACTS	2
1 EXECUTIVE OVERVIEW	3
1.1 INTRODUCTION	4
1.2 AUDIT SUMMARY	5
1.3 TEST APPROACH & METHODOLOGY	6
RISK METHODOLOGY	6
1.4 SCOPE	8
2 ASSESSMENT SUMMARY & FINDINGS OVERVIEW	9
3 FINDINGS & TECH DETAILS	10
3.1 (HAL-01) LACK OF THRESHOLD CHECK - MEDIUM	12
Description	12
Code Location	12
Example Functions	13
Risk Level	13
Recommendation	13
Remediation Plan	13
3.2 TESTING SIGNATORY/VALIDATOR WORKFLOWS	14
Description	14
Results	18
3.3 TESTING OUT-OF-ORDER	19
Description	19
Results	21

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	05/19/2021	Gabi Urrutia
0.2	Document Edits	05/28/2021	Gokberk Gulgun
1.0	Final Version	05/31/2021	Gokberk Gulgun
1.1	Remediation Plan	06/07/2021	Gokberk Gulgun

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Gokberk Gulgun	Halborn	Gokberk.Gulgun@halborn.com



EXECUTIVE OVERVIEW



1.1 INTRODUCTION

Yieldly.Finance bridge component is designed to integrate a diverse set of blockchains specialised for different needs. **Yieldly.Finance** connects Algorand to Ethereum , and vice versa.

Yieldly.Finance engaged Halborn to conduct a security assessment on their Algorand Smart contract beginning on May 19, 2021 and ending May 31th, 2021. The security assessment was scoped to the Algorand lottery contracts and an audit of the security risk and implications regarding the changes introduced by the development team at **Yieldly.Finance** prior to its production release shortly following the assessments deadline.

Though this security audit's outcome is satisfactory, only the most essential aspects were tested and verified to achieve objectives and deliverables set in the scope due to time and resource constraints. It is essential to note the use of the best practices for secure smart-contract development.

1.2 AUDIT SUMMARY

The team at Halborn was provided two weeks for the engagement and assigned three full time security engineers to audit the security of the smart contract. The security engineers are blockchain and smart-contract security experts with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

Risk Assessment Sheet

Risk Assessment	Status	Description
Access Control Policies Assessment	PASS	Authorization has been checked according to roles on functions.
Multi-Sig Assessment	PASS	It has been observed that multi-sig has been implemented in related contracts.
Decimal Calculation Assessment	PASS	In mathematical calculations, there is no problem that may cause overflow or unexpected calculations.
ReKeyTo Property Assessment	PASS	It has been observed that the ReKeyTo variable is implemented with Zeroaddress control on related contracts.
Input Validation Assessment	PASS	The Signatory and Validator functions are correctly implemented according to workflow.
Freeze/Clawback Address Assessment	PASS	"Yieldly.Finance" Team confirmed the assets don't have "freeze/clawback" addresses.
Proxy Assessment	PASS	"Yieldly.Finance" Team applied the necessary changes to communicate through the proxy.
Fee And Amount Check Assessment	PASS	Fee and Amount checks are applied in the contracts.
Pragma Version Assessment	PASS	Same Pragma version are used on contracts.
Group Size Validation Assessment	PASS	The group size variable has been checked at the beginning of the function statements.
Alerthub Setup Assessment	PASS	"Yieldly.Finance" Team will set up Alerthub on the mainnet.

The purpose of this audit to achieve the following:

- Ensure that smart contract functions are intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified few security risks, and recommends performing further testing to validate extended safety and correctness in context to the whole of contract. External threats, such as economic attacks, oracle attacks, and inter-contract functions and calls should be validated for expected logic and state.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose.
- Smart Contract manual code read and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions(**buildr**)
- Manual Assessment of use and safety for the critical Algorand variables and functions in scope to identify any arithmetic related vulnerability classes.
- Smart Contract Dynamic Analysis And Flow Testing

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident, and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. It's quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that was used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

- 5 - Almost certain an incident will occur.
- 4 - High probability of an incident occurring.
- 3 - Potential of a security incident in the long term.
- 2 - Low probability of an incident occurring.

1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.

4 - May cause a significant level of impact or loss.

3 - May cause a partial impact or loss to many.

2 - May cause temporary impact or loss.

1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
----------	------	--------	-----	---------------

10 - CRITICAL

9 - 8 - HIGH

7 - 6 - MEDIUM

5 - 4 - LOW

3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

Code related to [Yieldly Algorand Bridge Repository](#)

Specific commit of contract:

`14aed4bd123ff6b95e470a8ed993cdc11fa4b01a`

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	0	0

LIKELIHOOD

IMPACT

		(HAL-01)		

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LACK OF THRESHOLD CHECK	Medium	SOLVED: 06/02/2021
TESTING SIGNATORY/VALIDATOR WORKFLOWS	-	-
TESTING OUT-OF-ORDER	-	-



FINDINGS & TECH DETAILS



3.1 (HAL-01) LACK OF THRESHOLD CHECK – MEDIUM

Description:

In the `Yieldly.Finance`, Two type of role is defined in the bridge contracts. They are named as `Signatory` and `Validator`. Signatory and validator thresholds have been implemented in an editable way on the related functions. However, there is no limit on these functions.

Code Location:

Function `sigThresholdProp`

Listing 1: `HalbornTest.js` (Lines)

```
1 it("Halborn Should propose threshold should change to 20", async
  () => {
2   return await new Promise(async (resolve, reject) => {
3     try {
4       let txn = await configs.sigThresholdProp(account1, sigAppId,
5         20);
6       resolve(txn);
7     } catch (err) {
8       console.log(err);
9     }
10  });
11 }).timeout(120000);
```

Function `valThresholdProp`

Listing 2: `HalbornTest.js` (Lines)

```
1 it("Halborn Should propose threshold should change to 20", async
  () => {
2   return await new Promise(async (resolve, reject) => {
3     try {
4       let txn = await configs.valThresholdProp(account1, valAppId,
5         20, sigAppId);
```

```
5     resolve(txn);
6   } catch (err) {
7     reject();
8     assert.fail("Failed to propose");
9   }
10  });
11 }).timeout(120000);
```

Example Functions:

Listing 3

```
1 valThresholdProp
2 sigThresholdProp
```

Risk Level:

Likelihood - 3

Impact - 3

Recommendation:

The limit definition range of threshold should be defined in the contract functionalities.

Remediation Plan:

SOLVED: **Yieldly.Finance** Team defined threshold limit on the functions.

3.2 TESTING SIGNATORY/VALIDATOR WORKFLOWS

Description:

During the test process, Two type of user have been defined on the contracts. One of them is defined as signatory and other one named as validator. In the testing process, Functions accessible to relevant users have been checked. A signatory user functions are shown in the below.

Listing 4: Functions (Lines)

```
1 function isSignatory()
2 function sigThresholdProp()
3 function signatoryProp()
4 function signatoryApprove()
5 function
```

Next, privileged validator functions are extracted from the test cases and shown below.

Listing 5: Functions (Lines)

```
1 function isValidator()
2 function addValidatorProp()
3 function valThresholdProp()
```

All functions are tested through Mocha framework. Two accounts provided by Yieldly .Finance team and one account has been created by Halborn team. After importing accounts into Mocha and AlgoSDK, Signatory and Validator workflows are evaluated according to the following code parts. Tests are completed through Algorand Testnet.

Listing 6: IsValidator Check (Lines)

```
1 it("Halborn Test Validator Check - PASS", async () => {
2   return await new Promise(async (resolve, reject) => {
3     try {
```



```

4      let txn = await configs.isValidator(account1, valAppId);
5      resolve(txn);
6    } catch (err) {
7      reject();
8      assert.fail("Failed to propose");
9    }
10  });
11 }).timeout(120000);
12
13 it("Halborn Test Validator Check - FAIL", async () => {
14   return await new Promise(async (resolve, reject) => {
15     try {
16       let txn = await configs.isValidator(account2, valAppId);
17       reject(assert.fail("Should have failed"));
18     } catch (err) {
19       resolve();
20     }
21   });
22 }).timeout(120000);
23

```

Listing 7: Add Validatory and Approve Check (Lines)

```

1  it("Halborn Test Add Validator - PASS", async () => {
2    return await new Promise(async (resolve, reject) => {
3      try {
4        let txn = await configs.addValidatorProp(
5          account1,
6          valAppId,
7          account2,
8          sigAppId
9        );
10       resolve(txn);
11     } catch (err) {
12       reject();
13       assert.fail("Failed to propose");
14     }
15   });
16 }).timeout(120000);
17
18 it("Halborn Test Validator Approve", async () => {
19   return await new Promise(async (resolve, reject) => {
20     try {
21       let txn = await configs.validatorApprove(

```

```

22     account2,
23     valAppId,
24     account2,
25     sigAppId
26   );
27   resolve(txn);
28 } catch (err) {
29   reject();
30   assert.fail("Failed to propose");
31 }
32 });
33 }).timeout(120000);

```

Listing 8: Validator Threshold Change Check (Lines)

```

1 it("Halborn Added Validator Threshold Change Check", async () => {
2   return await new Promise(async (resolve, reject) => {
3     try {
4       let txn = await configs.valThresholdApprove(account2,
5         valAppId, sigAppId);
6       resolve(txn);
7     } catch (err) {
8       reject();
9       assert.fail("Failed to propose");
10    }
11  }).timeout(120000);

```

Listing 9: Signatory Check (Lines)

```

1 it("Halborn Signatory Check", async () => {
2   return await new Promise(async (resolve, reject) => {
3     try {
4       let txn = await configs.isSignatory(account3, sigAppId);
5       reject(assert.fail("Should have failed"));
6     } catch (err) {
7       resolve();
8     }
9   });
10 }).timeout(120000);
11
12 it("Halborn Add Signatory Check", async () => {
13   return await new Promise(async (resolve, reject) => {

```

```

14     try {
15         let txn = await configs.signatoryProp(account1, sigAppId,
            account2);
16         resolve(txn);
17     } catch (err) {
18         reject();
19         assert.fail("Failed to propose");
20     }
21 });
22 }).timeout(120000);

```

Listing 10: Signatory Threshold Change (Lines)

```

1 it("Halborn Signatory Threshold Change", async () => {
2     return await new Promise(async (resolve, reject) => {
3         try {
4             let txn = await configs.sigThresholdProp(account1, sigAppId,
                2);
5             resolve(txn);
6         } catch (err) {
7             reject();
8             assert.fail("Failed to propose");
9         }
10    });
11 }).timeout(120000);

```

By running the relevant codes on each function, the results were examined on [Testnet](#). As a result of the tests performed in a limited time, no problems were observed in the flows.

Transaction Overview

Transaction ID
X42LBWSRK55BFDY3HAV4QBR35K75M7XLDZ62DEQKX7BBLQNIUQ4Q

Timestamp
Tue May 25 2021 16:23:43 GMT-0400

Block
14374212

Type
Application Call

Status
Completed

Transaction Details

Group ID:	UTPRo/fu8hOa9lpjEWTvJJs9SC3D8xKmjdsJzuD3LW0=	Copy
Sender:	JC3J5M3KK5DGGRLBOFI4Q2IEQEWJ2ZL2PKJ36UPCIY4ND5XLQD7VX2HGEI	Copy
Application ID:	15951566	Copy
Application Version:	2	
On Completion:	Call	

Application args

aXNTaWduYXRvcnk=

Results:

According to an analysis, It has been observed that the transactions produced by the functions against workflow manipulation are as expected. The Function enhancements are structured with roles.

3.3 TESTING OUT-OF-ORDER

Description:

In the smart contracts, the grouped transactions are examined by changing their orders. The relevant changes are completed on the test cases.

Function valThresholdPropReverse

Listing 11: HalbornTest.js (Lines)

```

1 it("valThresholdPropReverse - Reverse", async () => {
2   return await new Promise(async (resolve, reject) => {
3     try {
4       let txn = await configs.valThresholdPropReverse(account1,
5         valAppId, 2, sigAppId);
6       resolve(txn);
7     } catch (err) {
8       console("ERROR Reverse valThresholdPropReverse");
9       console.log(err);
10    }
11  });
12 }).timeout(120000);

```

Function signatoryApproveReverse

Listing 12: HalbornTest.js (Lines)

```

1 it("signatoryApproveReverse - Reverse Orders", async () => {
2   return await new Promise(async (resolve, reject) => {
3     try {
4       let txn = await configs.signatoryApproveReverse(account2,
5         sigAppId, account3);
6       resolve(txn);
7     } catch (err) {
8       console.log("ERROR - REVERSE");
9     }
10  });
11 }).timeout(12000);

```

Function addValidatorPropReverse

Listing 13: HalbornTest.js (Lines)

```

1 it("addValidator Reverse Order", async () => {
2   return await new Promise(async (resolve, reject) => {
3     try {
4       let txn = await configs.addValidatorPropReverse(
5         account1,
6         valAppId,
7         account2,
8         sigAppId
9       );
10      resolve(txn);
11    } catch (err) {
12      console("ERROR Reverse addValidatorPropReverse");
13      console.log(err);
14    }
15  });
16 }).timeout(120000);

```

Function Reverse Group Example

Listing 14: Function Reverse Group Example (Lines)

```

1   var txngroup = await algodsk.assignGroupID([application,
2     verifier]);
3   application.group = txngroup[1].group;
4   verifier.group = txngroup[0].group;
5
6   var signed2 = await application.signTxn(account.sk);
7   var signed1 = await verifier.signTxn(account.sk);
8
9   var bytes = concatArrays(signed1, signed2);
10
11  var { txId: createTxId } = await algodClient

```

The tests were carried out in interaction with **Testnet** over the **Mocha**.

```

Error: Bad Request
    at Request.callback (/home/zilion/yieldly-bridge-smart-contracts-master/yieldly-bridge-algo/node_modules/superagent/lib/node/index.js:883:15)
    at /home/zilion/yieldly-bridge-smart-contracts-master/yieldly-bridge-algo/node_modules/superagent/lib/node/index.js:1127:28
    at IncomingMessage.<anonymous> (/home/zilion/yieldly-bridge-smart-contracts-master/yieldly-bridge-algo/node_modules/superagent/lib/node/parsers/json.js:22:7)
    at IncomingMessage.emit (events.js:327:22)
    at endReadableNT (_stream_readable.js:1220:12)
    at processTicksAndRejections (internal/process/task_queues.js:84:21) {
  status: 400,
  response: Response {
    _events: [Object: null prototype] {},
    _eventsCount: 0,
    _maxListeners: undefined,
    res: IncomingMessage {
      _readableState: [ReadableState],
      readable: false,
      _events: [Object: null prototype],
      _eventsCount: 4,
      _maxListeners: undefined,
      socket: [Socket],
      connection: [Socket],
      httpVersionMajor: 1,
      httpVersionMinor: 1,
      httpVersion: '1.1',
      complete: true,
      headers: [Object],
      rawHeaders: [Array],
      trailers: {},
      rawTrailers: [],
      aborted: false,
      upgrade: false,
      url: '',
      method: null,
      statusCode: 400,
      statusMessage: 'Bad Request',
      client: [Socket],
      _consumed: false,
      _dumped: false,
      req: [ClientRequest],
      text: '{"message":"TransactionPool.Remember: transaction GEKTE3GN3HFPANWFZHAS13TOUVUJ3QABEAB36R4DTECUVL5HQ73IQ: transaction rejected by ApprovalProgram"}\n',
      [Symbol(kCapture)]: false
    },
    request: Request {
      _events: [Object: null prototype],
      _eventsCount: 1,
      _maxListeners: undefined,
      _enableHttp: false,
      _agent: false,
      _formData: null,
      method: 'POST',
      url: 'http://13.55.58.32:8080/v2/transactions',
      _header: [Object],

```

Results:

As a result of the tests, Reversed orders are checked on the grouped transactions.



THANK YOU FOR CHOOSING

// HALBORN

