// HALBORN

Yieldly.Finance Bridge Algorand

Smart Contracts Security Audit

Prepared by: **Halborn**

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

Visit: **Halborn.com**

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.

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 which were solved by Yieldly.Finance team. The fixes have been reviewed by the auditors.

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 dont'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.

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.

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

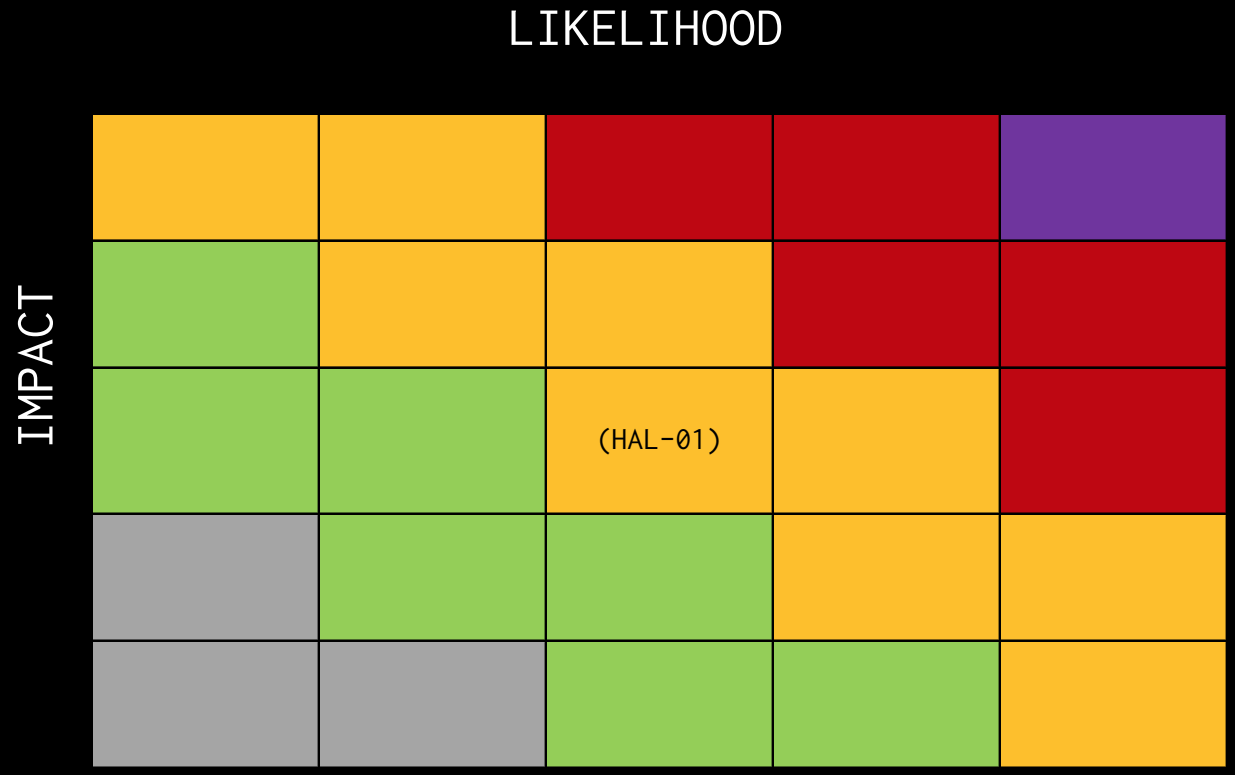
1.4 SCOPE

Code related to Yieldly Algorand Bridge Repository

Specific commit of contract:
d52a00210ab77ab5500aa159305725d15ae709a8

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	0	0



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LACK OF THRESHOLD CHECK	Medium	SOLVED: 06/02/2021
MANUAL TESTING	-	-

FINDINGS & TECH DETAILS

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

Description:

In the Yieldly.Finance, Three type of role is defined in the bridge contracts. They are named as Signatory and Validator and Dispatcher. 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,
            20);
5       resolve(txn);
6     } catch (err) {
7       console.log(err);
8     }
9   });
10 }).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,
```

```
            20, sigAppId);
      resolve(txn);
    } catch (err) {
      reject();
      assert.fail("Failed to propose");
    }
  });
}).timeout(120000);
```

Example Functions:

Listing 3

```
valThresholdProp
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.

MANUAL TESTING

During the manual testing multiple questions where considered while evaluation each of the defined functions:

- Can it be re-called changing admin/roles and permissions?
- Do we control sensitive or vulnerable parameters?
- Does the function check for boundaries on the parameters and internal values? Bigger than zero or equal? Argument count, array sizes, integer truncation.
- Can we bypass Proxy restrictions and interact with the escrow directly?
- Can we manipulate transaction order with re-ordering?
- Is there any missed address check?

4.1 TESTING SIGNATORY/VALIDATOR FUNCTIONS

ACCESS CONTROL CHECK:

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) => {
```

```
    try {
      let txn = await configs.addValidatorProp(
        account1,
        valAppId,
        account2,
        sigAppId
      );
      resolve(txn);
    } catch (err) {
      reject();
      assert.fail("Failed to propose");
    }
  });
}).timeout(120000);

it("Halborn Test Validator Approve", async () => {
  return await new Promise(async (resolve, reject) => {
    try {
      let txn = await configs.validatorApprove(
        account2,
        valAppId,
        account2,
        sigAppId
      );
      resolve(txn);
    } catch (err) {
      reject();
      assert.fail("Failed to propose");
    }
  });
}).timeout(120000);
```

Listing 8: Validator Threshold Change Check (Lines)

```
it("Halborn Added Validator Threshold Change Check", async () => {
  return await new Promise(async (resolve, reject) => {
    try {
      let txn = await configs.valThresholdApprove(account2,
          valAppId, sigAppId);
      resolve(txn);
    } catch (err) {
      reject();
      assert.fail("Failed to propose");
    }
```

```
  });
}).timeout(120000);
```

Listing 9: Signatory Check (Lines)

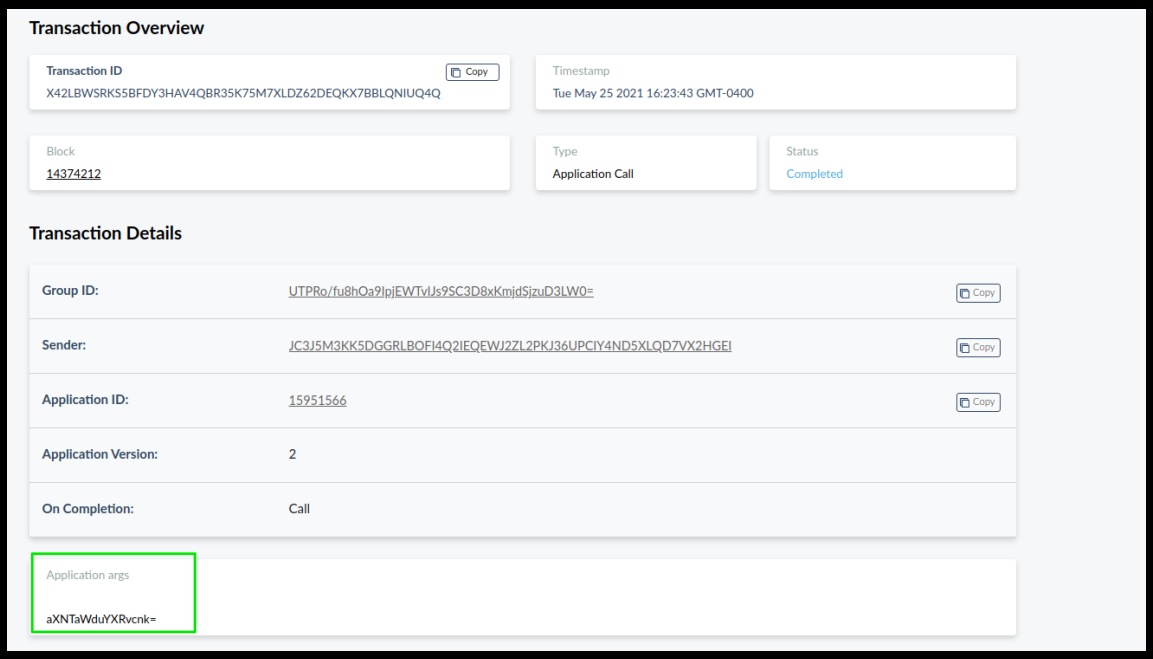
```
it("Halborn Signatory Check", async () => {
  return await new Promise(async (resolve, reject) => {
    try {
      let txn = await configs.isSignatory(account3, sigAppId);
      reject(assert.fail("Should have failed"));
    } catch (err) {
      resolve();
    }
  });
}).timeout(120000);

it("Halborn Add Signatory Check", async () => {
  return await new Promise(async (resolve, reject) => {
    try {
      let txn = await configs.signatoryProp(account1, sigAppId,
          account2);
      resolve(txn);
    } catch (err) {
      reject();
      assert.fail("Failed to propose");
    }
  });
}).timeout(120000);
```

Listing 10: Signatory Threshold Change (Lines)

```
it("Halborn Signatory Threshold Change", async () => {
  return await new Promise(async (resolve, reject) => {
    try {
      let txn = await configs.sigThresholdProp(account1, sigAppId,
          2);
      resolve(txn);
    } catch (err) {
      reject();
      assert.fail("Failed to propose");
    }
  });
}).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.



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.

INPUT VALIDATION CHECK:

During the test process, the signatory and validator functions have been reviewed by the auditors. In the testing process, Functions accessible to relevant users have been checked.

Without pragma version definition, The contract will be interpreted as a version 1 contract. In the contracts, the pragma version 3 used.

Bridge Signatory Contract

Listing 11: (Lines 1)

```
1 #pragma version 3
2
3
4 //** Begin
5 //** Descrption: Checks if not first time created, if so then
      initialise all global variables
6 //**/
7 int 0
8 txn ApplicationID
9 ==
10 bz not_creation
11 byte "Creator"
12 txn Sender
13 app_global_put
14 byte "Owner"
15 txn Sender
16 app_global_put
```

Bridge Validators Contract

Listing 12: (Lines 1)

```
1 #pragma version 3
2
3
4 //** Begin
5 //** Descrption: Checks if not first time created, if so then
      initialise all global variables
6 //**/
```

```
int 0
txn ApplicationID
==
bz not_creation
byte "Creator"
txn Sender
app_global_put
byte "Owner"
txn Sender
app_global_put
```

OUT OF ORDER CHECK:

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 13: HalbornTest.js (Lines)

```
it("valThresholdPropReverse - Reverse", async () => {
  return await new Promise(async (resolve, reject) => {
    try {
      let txn = await configs.valThresholdPropReverse(account1,
          valAppId, 2, sigAppId);
      resolve(txn);
    } catch (err) {
      console("ERROR Reverse valThresholdPropReverse");
      console.log(err);
    }
  });
}).timeout(120000);
```

Function signatoryApproveReverse

Listing 14: HalbornTest.js (Lines)

```
it("signatoryApproveReverse - Reverse Orders", async () => {
  return await new Promise(async (resolve, reject) => {
    try {
      let txn = await configs.signatoryApproveReverse(account2,
          sigAppId, account3);
      resolve(txn);
    } catch (err) {
      console.log("ERROR - REVERSE");
    }
  });
}).timeout(12000);
```

Function addValidatorPropReverse

Listing 15: HalbornTest.js (Lines)

```
it("addValidator Reverse Order", async () => {
  return await new Promise(async (resolve, reject) => {
    try {
      let txn = await configs.addValidatorPropReverse(
        account1,
        valAppId,
        account2,
        sigAppId
      );
      resolve(txn);
    } catch (err) {
      console("ERROR Reverse addValidatorPropReverse");
      console.log(err);
    }
  });
}).timeout(120000);
```

Function Reverse Group Example

Listing 16: Function Reverse Group Example (Lines)

```
      var txngroup = await algosdk.assignGroupID([application,
          verifier]);

```

```
        application.group = txngroup[1].group;
        verifier.group = txngroup[0].group;

        var signed2 = await application.signTxn(account.sk);
        var signed1 = await verifier.signTxn(account.sk);

        var bytes = concatArrays(signed1, signed2);

        var { txId: createTxId } = await algodClient
```

Testnet over the Mocha.

```
Error: Bad Request
    at Request.callback (/home/zilon/yieldly-bridge-smart-contracts-master/yieldly-bridge-algo/node_modules/superagent/lib/node/index.js:883:15)
    at /home/zilon/yieldly-bridge-smart-contracts-master/yieldly-bridge-algo/node_modules/superagent/lib/node/index.js:1127:20
    at IncomingMessage.<anonymous> (/home/zilon/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],
      _consuming: false,
      _dumped: false,
      req: [ClientRequest],
      text: '{"message":"TransactionPool.Remember: transaction GEKTE3GN3HFPANWFZWASI3TOUVUJQABEABJ6R4DTECUVL5MQ73IQ: transaction rejected by ApprovalProgram"}\n',
      [Symbol(kCapture)]: false
    },
    request: Request {
      _events: [Object: null prototype],
      _eventsCount: 1,
      _maxListeners: undefined,
      _enableHttp2: false,
      _agent: false,
      _formData: null,
      method: 'POST',
      url: 'http://13.55.58.32:8080/v2/transactions',
      _header: [Object],
```

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

4.2 TESTING DISPATCHER FUNCTIONS

ACCESS CONTROL CHECK:

The Dispatcher is is designed to integrate a diverse set of blockchains specialised for different needs. There are three role based on the components. The final test carried out through dispatcher role.

Listing 17: Bridge Dispatcher Teal (Lines)

```
514 it("Halborn - Dispatcher New Escrow", async () => {
515   .....
516
517   var appArgs = [];
518   appArgs.push(algosdk.decodeAddress(escrowAddress).publicKey);
519   appArgs.push(new Uint8Array(getInt64Bytes(proxyAppId)));
520   txnList.push(
521     configs.updateApplication(
522       account1,
523       optingAppId,
524       program6,
525       program2,
526       appArgs
527     )
528   );
529
530   /* Update the proxy checks */
531   var appArgs5 = [];
532   appArgs5.push(algosdk.decodeAddress(escrowAddress).publicKey);
533   appArgs5.push(new Uint8Array(getInt64Bytes(disAppId)));
534   appArgs5.push(new Uint8Array(getInt64Bytes(optingAppId)));
535   txnList.push(
536     configs.updateApplication(
537       account1,
538       proxyAppId,
539       program7,
540       program2,
541       appArgs5
542     )
543   );
544 ....
545   return new Promise((resolve) => resolve());
546 }).timeout(120000);
```

Listing 18: (Lines)

```
514 it("Halborn - Dispatch From Account 1 and Vote Account 2", async
      () => {
515   return await new Promise(async (resolve, reject) => {
516     try {
517       let txn = await configs.releaseTxnApprove(
518         account1,
519         disAppId,
520         escrowAddress,
521         account2,
522         valAppId,
523         assetId,
524         800000000,
525         proxyAppId
526       );
527       assert(!txn, "Should have failed");
528     } catch (err) {
529       resolve();
530     }
531   });
532 }).timeout(120000);
```

Listing 19: (Lines)

```
514 it("Halborn - Dispatch From Account 2 and Vote Account 3", async
      () => {
515   return await new Promise(async (resolve, reject) => {
516     try {
517       let txn = await configs.releaseTxnApproveNoTxn(
518         account2,
519         disAppId,
520         account3,
521         valAppId,
522         assetId,
523         proxyAppId
524       );
525       assert(!txn, "Should have failed");
526     } catch (err) {
527       resolve();
528     }
529   });
```

```
530 }).timeout(120000);
```

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.

INPUT VALIDATION CHECK:

During the test process, the dispatcher function has been reviewed by the auditors. In the testing process, Functions accessible to relevant users have been checked.

Without pragma version definition, The contract will be interpreted as a version 1 contract. In the dispatcher contract, the pragma version 3 used.

Bridge Dispatcher Teal

Listing 20: (Lines 1)

```
1 #pragma version 3
2
3
4 //** Begin
5 //** Descrption: Checks if not first time created, if so then
      initialise all global variables
6 //**/
7 int 0
8 txn ApplicationID
9 ==
10 bz not_creation
11 byte "Creator"
12 txn Sender
13 app_global_put
14 byte "Owner"
15 txn Sender
16 app_global_put
```

The Contract implementations should check GroupSize to make sure the size corresponds to the number of transactions the logic is expecting.

Function voteFin

Listing 21: Bridge Dispatcher Teal (Lines 515,516,517)

```
514 voteFin:
515 global GroupSize
516 int 3
517 ==
518 assert
519 int 1
520 return
521
522 failed:
523 int 0
524 return
525 finished:
526 int 1
527 return
528
```

According to test results, Group Size precondition checks are implemented over all contracts.

The contract code should verify that the RekeyTo property of any transaction is set to the ZeroAddress unless the contract is specifically involved in a rekeying operation.

Listing 22: Bridge Dispatcher Teal (Lines)

```
463 //** Function: sendTxn
464 //** Descrption:
465 //**/
466 sendTxn:
467 gtxn 0 ApplicationID
468 byte "G"
469 app_global_get
470 ==
471 assert
472
473 gtxn 3 TypeEnum
474 int 4
475 ==
```

```
476 assert
477
478 int 1
479 byte "SendAmount"
480 app_local_get
481 gtxn 3 AssetAmount
482 >=
483 assert
484
485 gtxn 2 RekeyTo
486 global ZeroAddress
487 ==
488 gtxn 3 RekeyTo
489 global ZeroAddress
490 ==
491 &&
492 assert
```

According to the static analysis results, necessary controls were applied on the ReKeyTo variables of contracts.

THANK YOU FOR CHOOSING

// HALBORN