Yieldly.Finance

Lottery Smart Contracts Security Audit

Prepared by: **Halborn**

Date of Engagement: **May 8th - 19th, 2021**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	05/18/2021	Gabi Urrutia
0.2	Document Edits	05/18/2021	Gokberk Gulgun
1.0	Final Version	05/19/2021	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 engaged Halborn to conduct a security assessment on their Smart contracts beginning on May 08, 2021 and ending May 19th, 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	\`Yieldly.Finance\` Team will monitor assets by a multi-signature address.
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 balance of the person has been checked in the flows of the functions.
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	\`Yieldly.Finance\` Team updated \`pragma\` version on the related 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

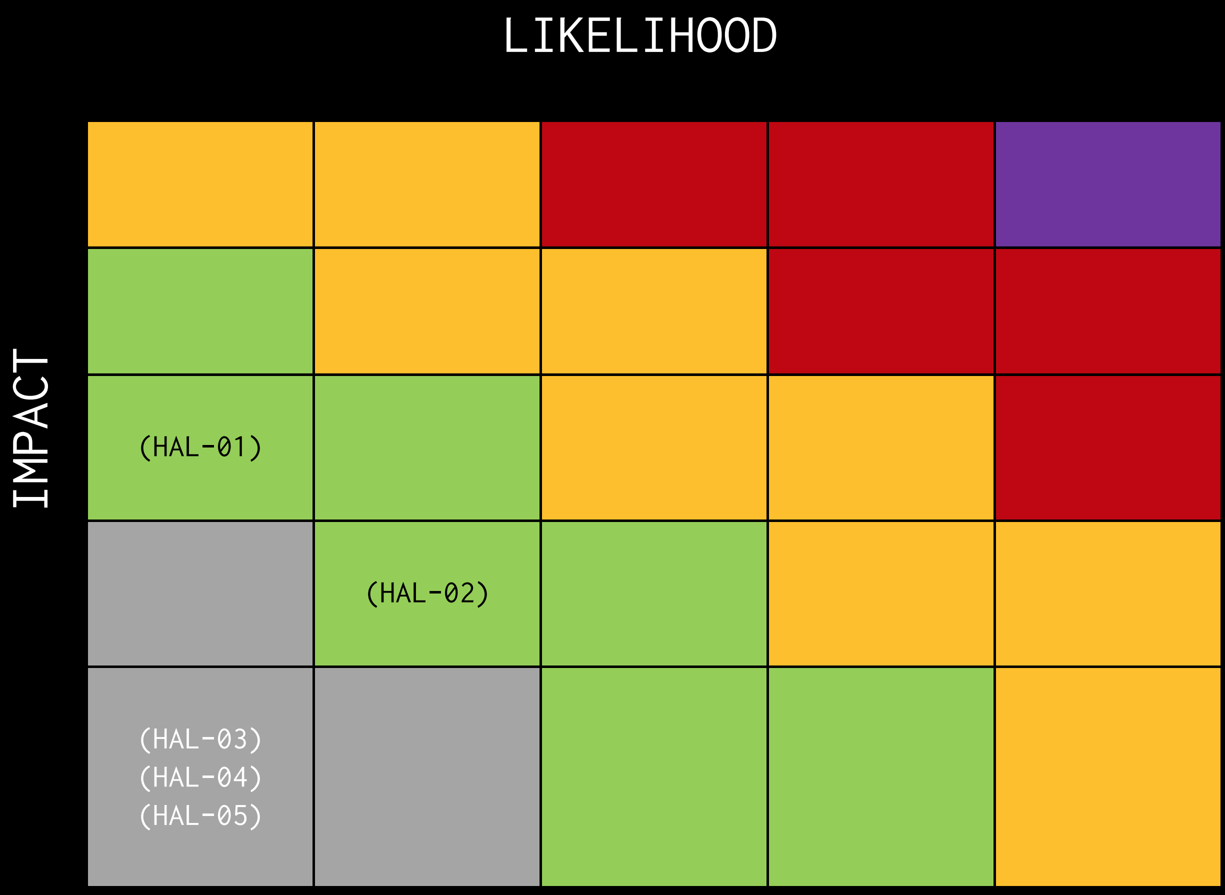
1.4 SCOPE

Code related to Yieldly Pools Repository

Specific commit of contract:
4bc5d8e49dfd8338306abcee91c7d5b44c114a09

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	2	3



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL01 - LACK OF MULTISIG PROGRAM	Low	SOLVED: 05/31/2021
HAL02 - MISSING PROXY ASSET DEFINITION ON THE FUNCTIONS	Low	SOLVED: 05/31/2021
HAL03 - MISSING FREEZE/REVOKE ASSETS DEFINITION	Informational	SOLVED: 05/31/2021
HAL04 - MULTIPLE PRAGMA DEFINITION	Informational	SOLVED: 05/31/2021
HAL05 - ALERTHUB SETUP	Informational	SOLVED: 05/31/2021
TESTING ACCESS CONTROL POLICIES	-	-
TESTING ALGORAND TEAL LANGUAGE IMPLEMENTATIONS	-	-
TESTING INPUT VALIDATION	-	-
TESTING OUT-OF-ORDER	-	-

FINDINGS & TECH DETAILS

3.1 (HAL-01) LACK OF MULTISIG PROGRAM - LOW

Description:

The principal benefit of multisig is that it creates added redundancy in key management. While single signature addresses require only a single key for transactions, multisignature addresses require multiple keys. To protect against malicious admin, it may be necessary to use a multi signature. By using this mechanism, a malicious admin actions could be prevented.

Code Location:

```
#!/bin/bash


date '+keyreg-teal-test start %Y%m%d_%H%M%S'

set -e
set -x
set -o pipefail
export SHELLOPTS

DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" >/dev/null 2>&1 && pwd )"

gcmd="goal"

ACCOUNT="JTCWA32ANVZBYN7JYR27QNJOAD52757NQ45EIAPAWOAN4Z4TXR2D3UDHZM"
#ACCOUNT="TNLDUGGEIMCWY5LVHGML6YZXC6DZGSZMUFADH72BA2D4CBU4KTKNUEPRH4"
WINNER="JTCWA32ANVZBYN7JYR27QNJOAD52757NQ45EIAPAWOAN4Z4TXR2D3UDHZM"

APPID="15788929"
APPID2="15788933"
APPID3="15788934"
APPID4="15788930"

ESCROW=$(${gcmd} clerk compile ../reward_fund_escrow.teal | awk '{ print $2 }'|tail -n 1)

${gcmd} app call --app-id $APPID --app-account=$ESCROW --app-account=$WINNER --app-arg "str:WN" --from=$ACCOUNT  --out=txn1.tx
${gcmd} app call --app-id $APPID2 --app-account=$ESCROW --app-arg "str:ATP" --foreign-app $APPID --from=$ACCOUNT  --out=txn2.tx
${gcmd} app call --app-id $APPID2 --app-account=$ESCROW --app-arg "str:UAT" --foreign-app $APPID --foreign-app $APPID4 --from=$ACCOUNT  --out=txn3.
${gcmd} app call --app-id $APPID --app-account=$ESCROW --app-arg "str:UAT" --foreign-app $APPID4 --from=$ACCOUNT  --out=txn4.tx
${gcmd} app call --app-id $APPID3 --app-arg "str:update" --from=$ACCOUNT  --out=txn5.tx


cat txn1.tx txn2.tx txn3.tx txn4.tx txn5.tx> combinedtxn.tx
${gcmd} clerk group -i combinedtxn.tx -o groupedtxn.tx
${gcmd} clerk sign -i groupedtxn.tx -o signout.tx
${gcmd} clerk rawsend -f signout.tx
#${gcmd} clerk dryrun -t signout.tx --dryrun-dump -o dump1.dr
#tealdbg debug ./reward_fund_test.teal -d dump1.dr

${gcmd} app read --app-id $APPID --guess-format --global --from $ACCOUNT
${gcmd} app read --app-id $APPID --guess-format --local --from $ACCOUNT

rm *.tx
```

Example Definition:

Listing 1: Multisig Implementation (Lines)

```
goal account multisig new -T 2 account1 account2 account3 -d ~/
    node/data
goal clerk multisig signprogram -p /tmp/*.teal -a account1 -A
    account2 -o /tmp/simple.lsig -d ~/node/data
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

In the contract, The multi-signature should be implemented over a creator account.

Remediation Plan:

SOLVED: Yieldly.Finance Team will monitor assets by a multi-signature address.

3.2 (HAL-02) MISSING PROXY ASSET DEFINITION ON THE FUNCTIONS - LOW

Description:

In the Yieldly.Finance workflow, Escrow connection is made with a proxy contract. According to documentation, Escrow only allows transactions tied with proxy. But, in the some of functions transactions don't go through the Proxy asset.

Code Location:

Listing 2: winnerProgram Function (Lines 1)

```
let txn = await configs.winnerProgram(
      account2,
      escrowAddress,
      algoAppId,
      asaAppId,
      trackerAppId,
      winner,
      rateAppId
  );
```

Listing 3: assetOptoutApplication Function (Lines 1)

```
let txn1 = await configs.assetOptoutApplication(
      account1,
      escrowAddress,
      optingAppId,
      assetId
  );
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

It is recommended to construct transactions through a proxy which is interacting with escrow.

Remediation Plan:

SOLVED: Yieldly.Finance Team applied the necessary changes to communicate through the proxy.

3.3 (HAL-03) MISSING FREEZE/REVOKE ASSETS DEFINITION - INFORMATIONAL

Description:

When an asset is created, the contract can provide a freeze address and a defaultfrozen state. If the defaultfrozen state is set to true the corresponding freeze address must issue unfreeze transactions, one per account, to allow trading of the asset to and from that account. This may be useful in situations that require holders of the asset to pass certain checks prior to ownership. (KYC/AML) The clawback address, if specified, is able to revoke the asset from any account and place them in any other account that has previously opted-in. This may be useful in situations where a holder of the asset breaches some set of terms that you established for that asset. You could issue a freeze transaction to investigate, and if you determine that they can no longer own the asset, you could revoke the assets.

Code Location:

Application Details

Application Version:	2	
Creator:	JTCWA32ANVZBYN7JYR27QNJOAD52757NQ45EIAPAWOAN4Z4TXR2D3UDHZM	Copy
Destroyed:	False	
numByteSlice:	8	
numUint:	8	

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

According to workflow, the application should activate freeze and revoke assets. If the application would rather ensure to asset holders that the application will never have the ability to revoke or freeze assets, set the clawback/freeze address to null.

SOLVED: Yieldly.Finance Team confirmed the assets dont't have freeze/clawback addresses.

3.4 (HAL-04) MULTIPLE PRAGMA DEFINITION - INFORMATIONAL

Description:

It has been observed that different versions of the pragma are used on TEAL contracts. The pragma on the ESCROW contract is defined as 2.

Code Location:

Listing 4: Pragma Version 2 Functions (Lines)

```
reward_fund_escrow.teal
reward_fund_close.teal
reward_fund_rates.teal
reward_fund_tracker.teal
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

A common version of pragma (3) should be used across all contracts to avoid an unexpected workflows.

SOLVED: Yieldly.Finance Team updated pragma version on the related contracts.

3.5 (HAL-05) ALERTHUB SETUP - INFORMATIONAL

Description:

AlertHub is a tool that provides monitoring and real-time alerts on Algorand addresses so that users may manage the security of their accounts and the wider Algorand network.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to setup alerthub for real-time monitoring. It can help the operations proceed healthily and safely.

SOLVED: Yieldly.Finance Team will set up Alerthub on the mainnet.

3.6 TESTING ACCESS CONTROL POLICIES

Description:

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

Listing 5: Non-privileged Functions (Lines)

```
function claimASATokens()
function optoutApplication()
function withdrawLottery()
function withdrawASATokens()
function calcRewardASA()
function claimStaking()
function stakeASATokens()
function claimASATokens()
function depositLottery()
function giveEscrowAlgos()
function lockTokensLottery()
```

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

Listing 6: Privileged Functions (Lines)

```
function changeUnlockRate()
function changeUnlockRatio()
function winnerProgram()
function deleteApplication()
function assetOptoutApplication()
```

All functions are tested through Mocha framework. Two accounts provided by Yieldly .Finance team and one account has been created by Halborn team.

```
HALBORN ACCESS CONTROL TEST
Admin Account : JTCWA32ANVZBYN7JYR27QNJOAD52757NQ45EIAPAWOAN4Z4TXR2D3UDHZM
Yieldly Account : HGM6WT6AV256W55S3GUPDGLHYYK2UI3Q6MVKXLXY2K56UXP6MPQUWDOS7U
Halborn Test Account : TQWE35S2KGQMXDJHM44XYKZQXBL2JPBJT2YPUKFO5D5LFVY4WP5FJRRCZE
```

After importing accounts into Mocha and AlgoSDK, Access control policies are evaluated according to the following code parts. Tests are completed through Algorand Testnet.

Listing 7: Access Control Check - Change Asa Unlock Rate (Lines)

```
it("Access Control Check - Change Asa Unlock Rate", async () => {
  try {
    await getData();
    let testAmount = 80;

    let txn = await configs.changeUnlockRate(account3, testAmount, rateAppId);

    await getData();

    return txn;
  } catch (err) {
    console.log(stateData)

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

Listing 8: Access Control Check - Asset OptOut Application (Lines)

```
it("Access Control Check - Asset OptOut Application", async () => {

  try {
    let txn1 = await configs.optoutApplication(account3, asaAppId);
    assert(txn1, "Done");
  } catch (err) {
  }
  try {
    let txn2 = await configs.optoutApplication(account3, algoAppId
```

```
        );
      assert(txn2, "Done");
    } catch (err) {
    }
    try {
      let txn3 = await configs.optoutApplication(account3,
        trackerAppId);
      assert(txn3, "Done");
    } catch (err) {
    }
    try {
      let txn1 = await configs.assetOptoutApplication(
        account3,
        escrowAddress,
        optingAppId,
        assetId
      );
      assert(txn1, "Done");
    } catch (err) {
    }
    return new Promise((resolve) => resolve());
  }).timeout(120000);
```

Listing 9: Access Control Check - Change Unlock Ratio (Lines)

```
it("Access Control Check - Change Unlock Ratio between ASA stakers
    and lottery", async () => {
  try {
    await getData();
    let testAmount = 80;

    let txn = await configs.changeUnlockRatio(account3, testAmount
      , rateAppId);

    await getData();

    stateDataPrevious = stateData;

    return txn;
  } catch (err) {
    console.log(stateData)

    throw err;
```

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

Listing 10: Access Control Check - Delete Application (Lines)

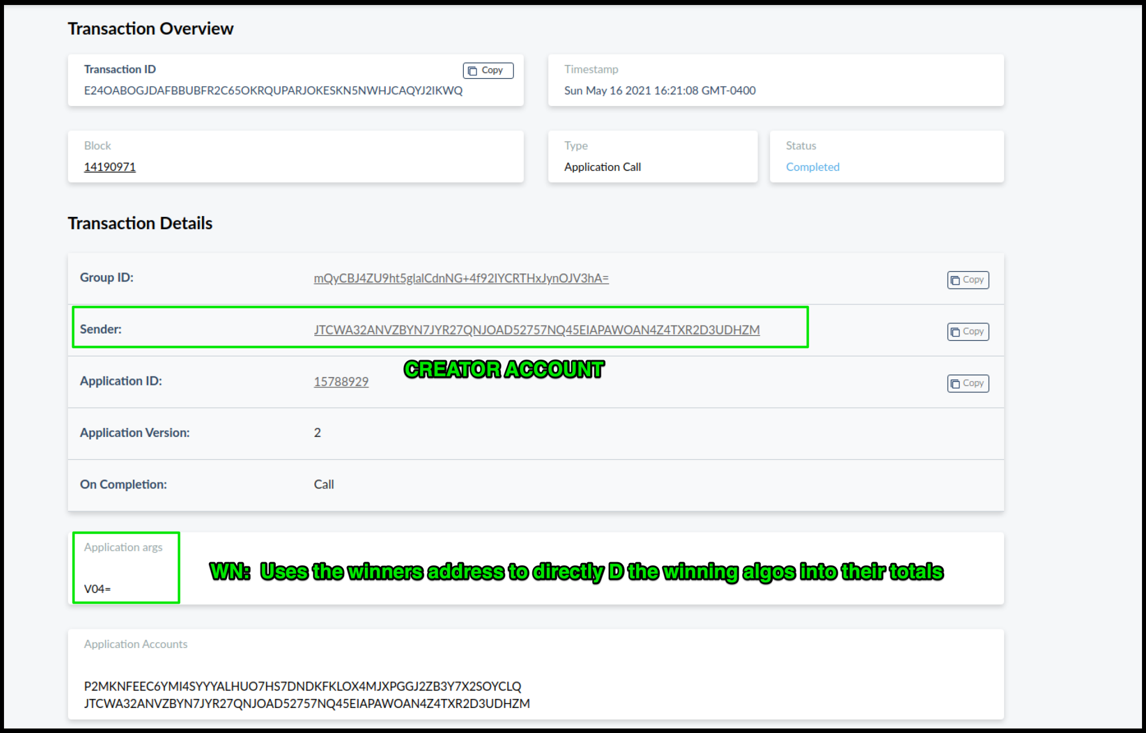
```
it("Should delete the application and clear assets for account 2",
    async () => {
  try {
    let txn2 = await configs.deleteApplication(
      account2,
      escrowAddress,
      algoAppId
    );
    assert(txn2, "Done");
  } catch (err) {
    console.log("Error on removing application 1")
  }
  try {
    let txn4 = await configs.deleteApplication(
      account2,
      escrowAddress,
      asaAppId
    );
    assert(txn4, "Done");
  } catch (err) {
    console.log("Error on removing application 2")
  }
  try {
    let txn6 = await configs.deleteApplication(
      account2,
      escrowAddress,
      optingAppId
    );
    assert(txn6, "Done");
  } catch (err) {
    console.log("Error on removing application 3")
  }
```

Listing 11: Access Control Check - Winner Program (Lines)

```
it("Access Control Check - Winner Program", async () => {
  try {
```

```
    await getData();

    let txn = await configs.winnerProgram(
      account1,
      escrowAddress,
      algoAppId,
      asaAppId,
      trackerAppId,
      winner,
      rateAppId
    );

    await getData();

    stateDataPrevious = stateData;

    return txn;
  } catch (err) {
    console.log(stateData);

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

Testnet. The functions policy has been checked whether it produces a transaction or not. The relevant example can be examined below.



Results:

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

3.7 TESTING ALGORAND TEAL LANGUAGE IMPLEMENTATIONS

ReKeyTo Verification:

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.

```
#pragma version 2

global GroupSize
int 2
>=
global GroupSize
int 6
<=
&&
bz failed

// Proxy Contract ID
gtxn 0 ApplicationID
//int 14737326
int 15858338
==
bnz resumes
b failed

resumes:
gtxn 0 TypeEnum
int 6
==

gtxn 0 OnCompletion
int NoOp
==
int DeleteApplication
gtxn 0 OnCompletion
==
||
&&

gtxn 1 RekeyTo
global ZeroAddress
==
&&
gtxn 0 RekeyTo
global ZeroAddress
==
&&
bnz success

failed:
int 0
return

success:
int 1
return
```

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

Fee & Amount Conditional Checks:

In the contracts, Fee and Amount conditional checks should be applied. During the analysis, all related TEAL contracts are checked. An unchecked Fee condition could burn the entire value of the contract account. Therefore, the necessity implementation should be completed at the beginning of statements.

```

//** Function: W
//** Descrption: Allows the users to W their algos from the lottery, their amoun
//**/
W:
// Safety checks
global GroupSize
int 4
==

// unnecessary, just let them send it to whomever they would like
// gtxn 2 Receiver
// gtxn 1 Sender
// ==

// Checks if amount + fee is less than amount user has deposited / won
gtxn 2 Amount
gtxn 2 Fee
+
int 0
byte "UA"
app_local_get
<=
&&
assert
```

With the analysis of Fee calculation, the conditional checks are applied on the related functions.

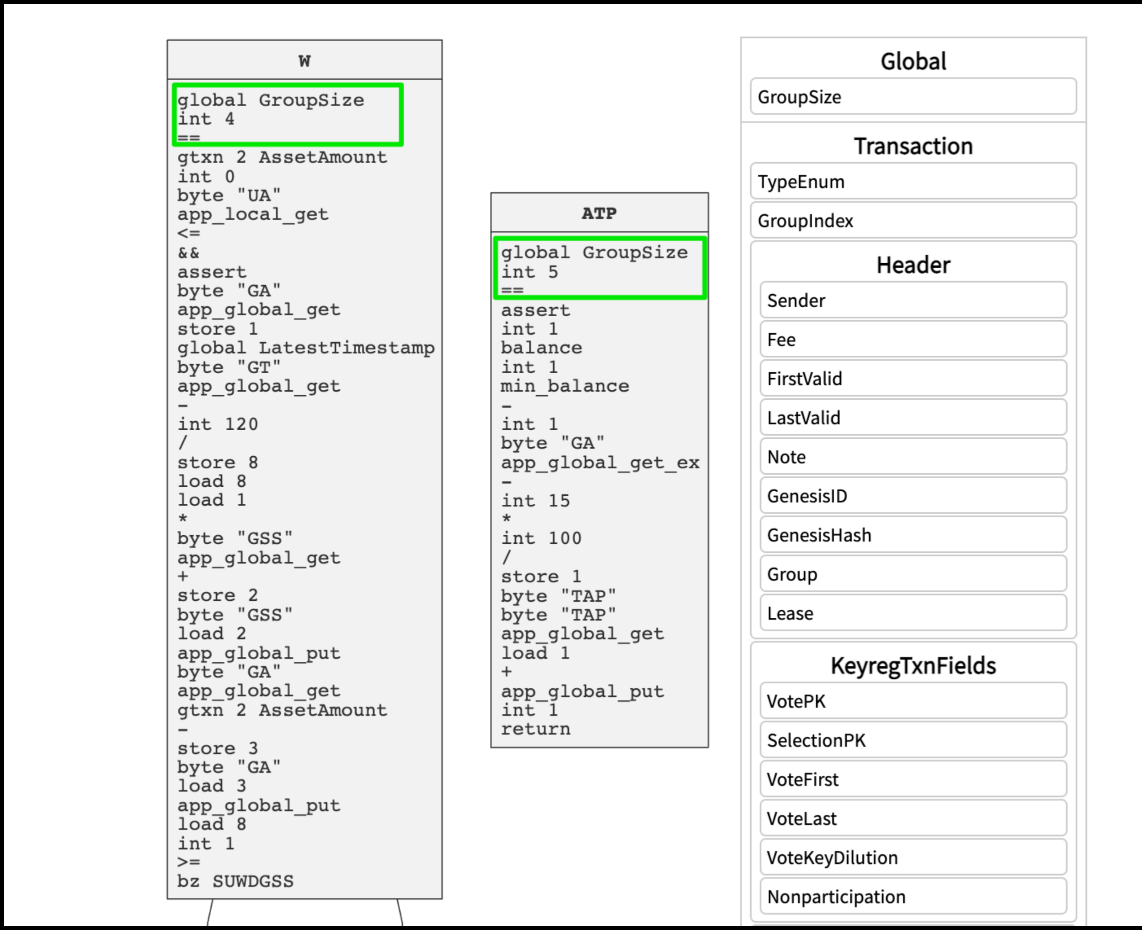
Pragma Version:

Without pragma version definition, The contract will be interpreted as a version 1 contract. In the lottery contracts, all pragma versions are defined.

```
#pragma version 3

//** Begin
//** Descrption: Checks if not first time created, if so then initialise all globa
//**/
int 0
txn ApplicationID
==
bz not_creation

// Sets creator as application creator
byte "C"
txn Sender
app_global_put

// Initialise the variables
byte "GT"
global LatestTimestamp
app_global_put
byte "GA"
int 0
app_global_put
byte "GSS"
int 0
app_global_put
```

GroupSize Check:

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



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

3.8 TESTING INPUT VALIDATION

Description:

In the smart contracts, the relevant tests have been carried out for functions using an user balance.

Listing 12: Input Validation - Deposit Lottery (Lines)

```
it("Input Validation - Deposit Lottery", async () => {
  try {
    await getData();
    let testAmount = stateData.account1.amountAlgo * 1000;

    let txn = await configs.depositLottery(
      account1,
      escrowAddress,
      testAmount,
      algoAppId,
      proxyAppId
    );

    return txn;
  } catch (err) {
    console.log(stateData)

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

Listing 13: Input Validation - Withdraw ASA Tokens (Lines)

```
it("Input Validation - Withdraw ASA Tokens", async () => {
  try {
    await getData();

    let txn = await configs.withdrawASATokens(
      account2,
      stateData.account2.asaStaking.UserAmount * 5000,
      escrowAddress,
      asaAppId,
      assetID,
```

```
      proxyAppId
    );

    await getData();
    return txn;
  } catch (err) {
    console.log(stateData)

    throw err;
  }
}).timeout(45000);
```

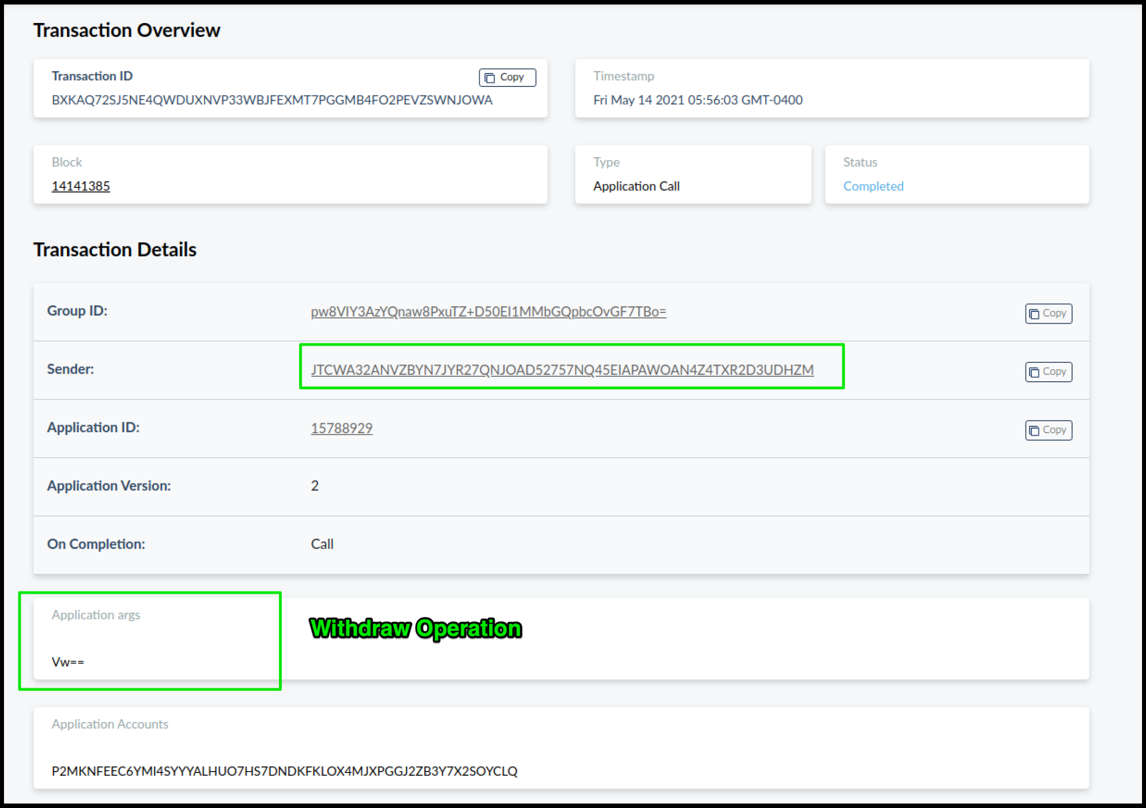
Listing 14: Input Validation - Lottery Withdraw (Lines)

```
it("Input Validation - Lottery Withdraw", async () => {
  try {
    await getData();

    let txn = await configs.withdrawLottery(
      account2,
      stateData.account2.lottery.UserAmount * 5000,
      escrowAddress,
      algoAppId,
      proxyAppId
    );
    await getData();

    return txn;
  } catch (err) {
    console.log(stateData)

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

Listing 15: Input Validation - Stake ASA Token (Lines)

```
it("Input Validation - Stake ASA Token", async () => {
  try {
    await getData();

    let tempAmount = stateData.account1.amountAsset * 1000000;

```

```
    let txn = await configs.stakeASATokens(
      account1,
      tempAmount,
      escrowAddress,
      asaAppId,
      assetID,
      proxyAppId
    );

    await getData();
    return txn;
  } catch (err) {
    console.log(stateData)

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



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

Sample transaction can be seen from the above.

Results:

As a result of the tests, User balances are checked on the deposit/withdraw/stake functions. Depending on the customer balance, Exceptional behaviours are handled and implementations have been put.

FINDINGS & TECH DETAILS

3.9 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 claimStakingReversed

Listing 16: HalbornTest.js (Lines)

```
exports.claimStakingReversed = (
  account,
  amount,
  amountAsa,
  escrow,
  appId,
  assetId,
  proxyId
) => {
  return new Promise(async (resolve, reject) => {
  ....
      proxyCheck.group = txngroup[0].group;
      applicationAsset.group = txngroup[2].group;
      application.group = txngroup[1].group;
      paymentAsset.group = txngroup[4].group;
      payment.group = txngroup[3].group;
      paymentEscrow.group = txngroup[5].group;

      var signed1 = await proxyCheck.signTxn(account.sk);
      var signed3 = await applicationAsset.signTxn(account.sk);
      var signed2 = await application.signTxn(account.sk);
      var signed4 = await algosdk.signLogicSigTransactionObject(
        txngroup[3],
        lsig
      );
      var signed5 = await algosdk.signLogicSigTransactionObject(
        txngroup[4],
        lsig
      );
  ....
    }
  });
```

```
};
```

Function stakeAlgoTokensReverse

Listing 17: HalbornTest.js (Lines)

```
exports.stakeAlgoTokensReverse = (account, amount, escrow, appId,
    proxyId) => {
  return new Promise(async (resolve, reject) => {
    try {
      /* opt vault into ASA staking */
      txParams = await algodClient.getTransactionParams().do();

       proxyCheck.group = txngroup[0].group;
      application.group = txngroup[2].group;
      payment.group = txngroup[1].group;

      var signed1 = await proxyCheck.signTxn(account.sk);
      var signed3 = await application.signTxn(account.sk);
      var signed2 = await payment.signTxn(account.sk);
    }
  });
};
```

Function withdrawASATokensReverse

Listing 18: HalbornTest.js (Lines)

```
exports.withdrawASATokensReverse = (
  account,
  amount,
  escrow,
  appId,
  assetId,
  proxyId
) => {
  return new Promise(async (resolve, reject) => {
    /* double check this after */
    try {
      txParams = await algodClient.getTransactionParams().do();
      ...
       proxyCheck.group = txngroup[0].group;
```

```
      application.group = txngroup[2].group;
      payment.group = txngroup[1].group;
      paymentEscrow.group = txngroup[3].group;

      var signed1 = await proxyCheck.signTxn(account.sk);
      var signed2 = await application.signTxn(account.sk);
      var signed3 = await algosdk.signLogicSigTransactionObject(
        txngroup[2],
        lsig
      );
      ....
    }
  });
};
```


```
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],
```


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

Results:

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

THANK YOU FOR CHOOSING

HALBORN