·Decurity·

# PROTOCOL SECURITY AUDIT REPORT

## EAST FINANCE

Decurity, 2021

CONTENTS

# 1 GENERAL INFORMATION

This report contains information about the results of the enterprise algorithmic stable token, conducted by Decurity (https://defisecurity.io/) in the period from 07/24/21 to 09/22/21.

## 1.1 Introduction

Tasks solved during the work are:

- Review of the EAST Token architecture and security design,
- Review of the contracts and service implementation,
- Description of the possible attack vectors and weaknesses of the system.

## 1.2 Scope of Work

The analyzed token and oracle contracts and the service source code are located in the private repositories. The commit hashes for the final analyzed versions of the repositories are:

- east-contract (TypeScript) – c024e87d46e84af88c2315601ea16e4b0138f215,
- east-service (TypeScript) – f3c85e2aafe1e71ebcdb4dfdcc107a83da39f3ac,
- oracle-contract (Scala) – 1577fd1c1b9974e7adbfa139bb3468ca5ac9ae13.

## 1.3 Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, token owner, token service owner, a contract).

The main possible threat actors are:

- User,
- Contract owner,
- Service owner,
- Oracle owners.

The table below contains sample attacks that malicious attackers might carry out.

Table 1. Possible attacks

| Attack | Actor |
|---|---|
| Contract code or data hijacking<br>*Deploying a malicious contract or submitting malicious data* | Contract owner<br>Oracle owner<br>Service owner |
| Financial fraud<br>*A malicious manipulation of the business logic and balances, such as a double-spending attack* | Anyone |
| Unauthorized transactions<br>*Neutralization of the access control leading to unauthorized messages accepted* | Anyone |
| Attacks on implementation<br>*Exploiting the weaknesses in the compiler or the runtime of the smart contracts* | Anyone |
| Attacks on off-chain service | Anyone |

| Attack | Actor |
| --- | --- |
| *Exploiting the weaknesses in the compiler or the runtime of the smart contracts* | |

## 1.4 Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation and severity (for the considered threats).

## 2 SUMMARY

As a result of this work, we have analyzed the contract and service interfaces, data flow, and access model. As a result, we've discovered several weaknesses and potential issues.

The EAST team has deployed the fixes for all the weaknesses and started implementing the best practices to avoid other potential issues.

### 2.1 Weaknesses

The table below contains the discovered issues, their risk level, and their status as of 09/27/2021.

Table 2. Discovered weaknesses

| Issue | Risk Level | Probability | Status |
|---|---|---|---|
| EAST funds can be irretrievably transferred to a non-existing vault | Medium | Medium | Fixed |
| The liquidate() and claimOverpay() methods don't check if the oracle rates are up to date | Medium | Low | Fixed |
| The close() method could be called twice with the same RWA/WEST transactions | Medium | Low | Fixed |

## 2.2 Potential issues

The table below contains the discovered issues, their risk level, and their status as of 09/27/2021.

Table 3. Discovered weaknesses

| Issue | Risk Level | Probability | Status |
|---|---|---|---|
| No guarantee for the user that the close_init() operation will be followed with an actual close | Medium | Low | Resolved |
| The close() method can fail if EAST funds are transferred following close_init() | Medium | Low | Resolved |
| Malicious contract owner can rebind the oracle contract to any address | Medium | Low | In Progress |
| Oracle rates are set off-chain, with no possibility of confirmation | Medium | Low | In Progress |

# 3 GENERAL RECOMMENDATIONS

This section contains general recommendations how to fix discovered during the testing weaknesses and vulnerabilities and how to improve overall security level.

Section 3.1 contains a list of general mitigations against the discovered weaknesses, technical recommendations for each finding can be found in section 4.

Section 3.2 describes a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level.

## 3.1 Current weaknesses remediation

- Fix the issues described in sections 4.1 and 4.2 .
- Perform review and verification of all the business scenarios of the contracts,
- Test the production environment of the off-chain service when deployed.

## 3.2 Security process improvement

- Rewrite the whitepaper to make it consistent with the implementation and the intended use cases of the system,
- Improve the decentralization by adding the timelock feature and oracle data validation,
- Launch a public bug bounty campaign for the contracts.

# 4 DETAILS

## 4.1 Discovered Issues

This section contains a brief summary of the issues which potentially might have been exploited by a malicious party.

### *4.1.1 [Fixed] EAST funds can be irretrievably transferred to a non-existing vault*

**Summary:**

In the `transfer` method of the EAST contract (**services/RPCService.ts**), the `to` parameter isn't validated. Supplying an incorrect address would make the contract transfer the EAST funds into a vault which would be inaccessible later.

**Remediation:**

We recommend adding a validation check to confirm that the vault corresponding to the `to` parameter exists.

**Retest:**

During the recheck, we've confirmed the fix deployed in **services/RPCService.ts**:

```
  async transfer(tx: Transaction, value: TransferParam):
Promise<DataEntryRequest[]> {
    await this.validate(TransferDto, value)
    const { to, amount: _amount } = value

    if(!this.isAddressValid(to)) {
      throw new Error(`Invalid transfer target address: '${to}'`);
```

```
  }
```

### *4.1.2 [Fixed] The liquidate() and claimOverpay() methods don't check if the oracle rates are up to date*

**Summary:**

The `liquidate` and `claimOverpay` methods of the EAST contract fetch the WEST rates without checking the timestamps.

**Remediation:**

We recommend using the `getLastOracles` function as it's done in other methods.

**Retest:**

During the recheck, we've confirmed the fix deployed in **services/RPCService.ts**:

```
  async liquidate(tx: Transaction, param: LiquidateParam):
Promise<DataEntryRequest[]> {
    await this.validate(LiquidateDto, param);
    . . .
    const { westRate } = await this.getLastOracles(oracleTimestampMaxDiff,
oracleContractId)


. . .


  async claimOverpay(tx: Transaction, param: ClaimOverpayParam) {
    await this.validate(ClaimOverpayDto, param)
    . . .
    const { rwaRate, westRate } = await this.getLastOracles(oracleTimestampMaxDiff,
oracleContractId);
```

### *4.1.3 [Fixed] The close() method could be called twice with the same RWA/WEST transactions*

**Summary:**

As the contract does not check the RWA/WEST transactions supplying the close operation for uniqueness, a malicious admin can close a vault reopened with the same supply without returning any funds to the user.

**Remediation:**

We recommend adding a check confirming that the RWA/WEST transfers haven't already been used in a `close`.

**Retest:**

During the recheck, we've confirmed the fix deployed in **services/RPCService.ts**:

```
async close(tx: Transaction, param: CloseParam): Promise<DataEntryRequest[]> {
  await this.validate(CloseDto, param)
  // only contract creator allowed
  const { address, rwaTransferId, westTransferId } = param
  . . .
    const isTransferUsedForClose = await
this.stateService.isTransferWestUsedForClose(westTransferId);
    if (isTransferUsedForClose) {
      throw new Error(`Transfer WEST '${westTransferId}' is already used for close`);
    }
  . . .
    const isTransferUsedForClose = await
this.stateService.isTransferRwaUsedForClose(rwaTransferId);
    if (isTransferUsedForClose) {
      throw new Error(`Transfer RWA '${rwaTransferId}' is already used for close`);
    }
```

```
. . .
if (westTransferId) {
  stateKeys.push({
    key: `${StateKeys.usedCloseWestTransfer}_${westTransferId}`,
    bool_value: true
  })
}

if (rwaTransferId) {
  stateKeys.push({
    key: `${StateKeys.usedCloseRwaTransfer}_${rwaTransferId}`,
    bool_value: true
  })
}
```

## 4.2 Potential Issues

This section contains a list of potential best practices violations which could lead to the vulnerabilities.

### *4.2.1 [Resolved] No guarantee for the user that the close_init() operation will be followed with an actual close*

**Summary:**

The EAST contract itself cannot guarantee that the users would be able to initiate the closing operation, as its actual execution requires trusting the external off-chain service following the blockchain events.

**Remediation:**

We recommend setting a flag within a contract confirming the close request, which would at least put an expectation that the operation would be followed through at some point. The flag should also block any operations over the user's vault except for `close` (which could fail, removing the flag).

**Retest:**

During the recheck, we've confirmed the fix deployed in **services/RPCService.ts**:

```
async handleDockerCall(tx: Transaction): Promise<void> {
  const { params } = tx;
  let results: DataEntryRequest[] = [];

  const param = params[0] || {};
  if (param) {
    if (Object.keys(Operations).filter(operationName => operationName !==
Operations.update_config).includes(param.key)) {
      await this.checkIsContractEnabled();
    }
    if ([
      Operations.reissue,
      Operations.close_init,
      Operations.claim_overpay_init,
      Operations.supply,
      Operations.liquidate
    ].includes(param.key as Operations)) {
      await this.checkVaultBlock(tx.sender);
    }

. . .
```

```
async mint(tx: Transaction, param: MintParam): Promise<DataEntryRequest[]> {
  . . .
  vault.isBlocked = false;

. . .

async closeInit(tx: Transaction): Promise<DataEntryRequest[]> {
  . . .
  vault.isBlocked = true;
```

#### *4.2.2 [Resolved] The close() method can fail if EAST funds are transferred following close_init()*

**Summary:**

The current off-chain admin service implementation does transfer closing funds and calls the contract's `close` method in an atomic transaction.

However, the contract itself doesn't require it, which leads to a possible situation when a user can transfer any amount of EAST out of the vault before the call to `close`. In that case, the call will fail the balance check, but the user will still receive the WEST/RWA tokens.

Such an attack could only be carried out if a vulnerable admin service implementation is deployed instead of the current one.

**Remediation:**

We recommend the `closing` flag preventing the transfer.

**Retest:**

During the recheck, we've confirmed the fix deployed in **services/RPCService.ts**. The fix is the same as for the previous issue.

### *4.2.3 Malicious contract owner can rebind the oracle contract to any address*

**Summary:**

As the contract owner can change the oracle addresses at any time via the `update_config` method, a malicious admin can use that to make the address point at any contract providing the required state values (or to an invalid address, making the contract inoperable).

A malicious contract owner could set the WEST rate to 0 and clean a vault leaving the users uncompensated during the `liquidate` call.

**Remediation:**

We recommend making the procedure of changing oracles (and possibly other parts of the configuration) two-step, effectively adding a timelock. With the timelock enabled, the admin would have to call `update_config_init` with the new parameters, then call `update_config_finalize` to confirm them but not before a specific time has passed.

We also recommend adding a more strict validation of the oracle addresses when setting the configuration to prevent choosing an invalid address for the oracle contract.

**Retest:**

The EAST team confirmed that they added the timelock feature in the EAST 1.1 roadmap and implemented a hardened environment to store the contract owner's private key.

#### *4.2.4 Oracle rates are set off-chain, with no possibility of confirmation*

**Summary:**

The oracle contract gets the rate values from a managing off-chain script without confirmation from the actual exchanges.

**Remediation:**

Since it's impossible to guarantee the correct exchange rates, we recommend the following possible solutions:

a) Add the sanity checks for the rates changes, e. g. detect too big changes.;
b) Implement synchronized on-chain rates fetching, e. g. request the rates at a particular moment modulo X seconds.

**Retest:**

The EAST team confirmed that they added the remediation to their roadmap.

# 5 APPENDIX

## 5.1 About us

The Decurity (former DeFiSecurity.io) team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained an expertise in blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.