TAGMI

Security Assessment
31st March 2023

Prepared for:
Petros Sideris

Prepared by:
Rome Rogers

1. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

2. Overview

TAGMI lets users create a wager on the outcome of any event, and tokenize the positions on either side of the bet as tradeable bearer assets. Using TAGMI, any two parties can create a bet on the outcome of any given event and designate an arbitrator that will decide who won the bet. The two parties can bet any ERC-20 tokens, in any amounts, and do not have to bet the same tokens or the same amounts against one another.

TAGMI will then issue NFTs to the two betting parties that represent their stakes in the wager. These NFTs are what we refer to as TAGs (short for Tokenized wAGers). These TAGs can be bought/sold/transferred to any other wallet. When a new party takes ownership of a TAG, they will receive payment in the event their position in the bet wins.

3. Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic and reputation damage of a successful attack
Likelihood - the chance that a particular vulnerability gets discovered and exploited
Severity - the overall criticality of the risk

4. Security assessment summary

review commit hash 1 - c3f9b38ddc712ba6d97d974e558c50186ab42d47
review commit hash 2 - 36b2aeff5392cb74b8b4449370056d9351df9f38

The following smart contracts were in scope of the audit:

```
TagEscrow
TagNFT
TagEscrowOtc
interfaces/**
```

Initially, I ran the contracts through MythX, which resulted in 1 low vulnerability being detected, included below. I then pursued a comprehensive deep dive. The following number of issues were found, categorised by their severity:

High: 3 issues
Medium: 3 issues
Low: 5 issues
Informational: 4 issues

5. Findings summary

ID	Title	Severity
[H-01]	Funds locked in escrow on `changeTagFee`	High
[H-02]	Funds locked in escrow due to unexpected ERC20 functionality	High
[H-03]	Incentive for arbitrator to misbehave	High
[M-01]	Lack of event emission after sensitive actions	Medium
[M-02]	Lack of validation when setting `determineTime`	Medium
[M-03]	Not following checks-effects-interactions pattern	Medium
[L-01]	Unused Event Logs	Low
[L-02]	Lack of indexed parameters in events	Low
[L-03]	Named return variables	Low
[L-04]	Missing error message in `require` statement	Low
[L-05]	`ArbitratorFeeBps = 0` (unnecessary gas)	Low
[I-01]	Improve inline documentation for getter functions	Informational
[I-02]	Unspecific compiler version `pragma`	Informational
[I-03]	Manual setting of NFT contract address	Informational
[I-04]	Use of `uint` instead of `uint256`	Informational

6. Findings

[H-01] Funds locked in escrow on `changeTagFee`

Severity

Impact: High, because any escrow that has been created or is currently running could potentially have its funds locked if there isn't enough of the relevant token in `TagEscrow`.

Likelihood: Medium, because it may happen for created and started escrows anytime `changeTagFee` is reduced.

Description

In `createEscrow`, the amount of an ERC20 to be transferred from Party A to `TagEscrow` is calculated by the amount minus the `tagVaultFee`.

```
if (address(currencyToDepositA) != address(0)) {
    currencyToDepositA.safeTransferFrom(msg.sender, address(this), 
amountToDepositA - amountToDepositA * tagFeeBps / maxFeeBps);
    currencyToDepositA.safeTransferFrom(msg.sender, tagFeeVault, 
amountToDepositA * tagFeeBps / maxFeeBps);
}
```

The same logic occurs for Party B when calling `depositFunds`.

When the escrow is ready for completion and the arbitrator calls `determineOutcome`, the amount of an ERC20 to be transferred to the winner from `TagEscrow` is calculated by the amount minus the `tagVaultFee`.

```
uint256 arbitratorFeeBps = escrows[escrowId].arbitratorFeeBps;
uint256 amountMinusFees = asset.amount - asset.amount * 
arbitratorFeeBps / maxFeeBps - asset.amount * tagFeeBps / maxFeeBps;
if (address(asset.currency) != address(0)) {
    asset.currency.safeTransfer(winner, amountMinusFees);
}
```

If the owner of `TagEscrow` was to update `changeTagFee` to a lower amount, any escrow between the states of `createEscrow` and `determineOutcome` may have their funds locked.

To test this vulnerability, I updated the following files:

```
test/functions/TagEscrow.ts
test/unit/TagEscrow.withdrawFunds.test.ts
```

In `TagEscrow.ts` I created a `changeTagFee` function that can be imported into our tests:

```
const changeTagFee = async (
  contract: ethers.Contract,
  signer: ethers.Signer,
  newFee: number,
) => {
  const tx = await contract
    .connect(signer)
    .changeTagFee(newFee);
  return await tx.wait()
}
```

In `TagEscrow.withdrawFunds.test.ts` I created two tests:

(1)

```
it('When fee is reduced after the escrow has started', async () => {
    await time.increase(3601);
    await escrowContract.changeTagFee(10);
    await expect(
    determineOutcome(escrowContract, partyArbitrator, escrowId,
false)
    ).to.be.revertedWith('ERC20: transfer amount exceeds balance');
});
```

(2) Note: I built a new describe block where I removed the `depositFunds` call in `beforeEach`.

```
 describe('Reverts2', () => {
    beforeEach(async function () {
        escrowId = await escrowContract.nextEscrowId();
        await approveTokenSpend(token, partyA,
escrowContract.address, numToEth(amount));
        await approveTokenSpend(token2, partyB,
escrowContract.address, numToEth(amount));
        await createEscrow(
            escrowContract,
            partyA,
            partyB.address,
            partyArbitrator.address,
            arbitratorFeeBps,
            'test',
            resolveTime,
            token.address,
            numToEth(100),
            token2.address,
            numToEth(100)
        );
        if (this.currentTest.title === 'When partyB has not deposited
any funds') return;
    });

    it('When fee is reduced before the escrow has started', async ()
=> {
        await escrowContract.changeTagFee(10);
        await depositFunds(
            escrowContract,
            partyB,
            escrowId,
        );
        await time.increase(3601);
        await expect(
            determineOutcome(escrowContract, partyArbitrator,
escrowId, false)
        ).to.be.revertedWith('ERC20: transfer amount exceeds
balance');
    });
});
```

Running `npx hardhat test test/unit/TagEscrow.determineOutcome.test.js` results in:

```
  TagEscrow.determineOutcome()
        ....
    Reverts
        ....
      ✓ When fee is reduced after the escrow has started
    Reverts2
      ✓ When fee is reduced before the escrow has started
```

This shows that when the fee is reduced, for all created escrows that have not been determined, calling `determineFunds` will result in an ERC20: transfer amount exceeds balance revert. Note that this just tests for `determineOutcome`, however it will be a problem in any function that attempts to send funds to a party.

Recommendations

Save the value of initial fee in `EscrowVault` and use that value when calculating how much needs to be sent to the winning party, instead of relying on the global variable `tagFeeBps`.

[H-02] Funds locked in escrow due to unexpected ERC20 functionality

Severity

Impact: High, because any escrow that uses an ERC20 with unexpected functionality could potentially have its funds locked if there isn't enough of the relevant token in `TagEscrow`.

Likelihood: Medium, because there is no ERC20 compliance whitelist.

Description

There are many well known ERC20 tokens that have unexpected functionalities that don’t comply with the standard. As TAGMI intends to allow any ERC20 token, I have focused on the instance where a user bets with a token that takes a fee on transfer.

To test this vulnerability, I updated the following files:

```
FakeFeeToken.sol
TagEscrow.determineOutcome.FakeFeeToken.test.js
```

In `FakeFeeToken.sol` I created a `_transfer` function that takes a percentage fee on transfer:

```
function _transfer(
    address sender,
    address recipient,
    uint256 amount
) internal override {
    uint256 fee = amount.mul(transferFeePercentage).div(100);
    uint256 netAmount = amount.sub(fee);
    super._transfer(sender, recipient, netAmount);
    super._transfer(sender, address(this), fee);
}
```

In `TagEscrow.determineOutcome.FakeFeeToken.test.js` I copied `TagEscrow.determineOutcome`, replaced `token2` with `feeToken` and ran a test to check for `determineOutcome` reverting when determining for Party A to win. I repeat this to confirm that no fees are sent to the arbitrator.

```
    describe('Reverts' , () => {
        let partyArbitratorToken1Balance,
            partyArbitratorToken2Balance;
        beforeEach(async () => {
            escrowId = await escrowContract.nextEscrowId();
            await approveTokenSpend(token, partyA,
escrowContract.address, numToEth(amount));
            await approveTokenSpend(feeToken, partyB,
escrowContract.address, numToEth(amount));
        });

        it('When determining for partyA', async () => {
            await createEscrow(
              escrowContract,
              partyA,
              partyB.address,
              partyArbitrator.address,
              arbitratorFeeBps,
              'test',
              resolveTime,
              token.address,
              numToEth(amount),
              feeToken.address,
              numToEth(amount)
            );
            await depositFunds(escrowContract, partyB, escrowId);
            await time.increase(3601);
            await expect(
                determineOutcome(escrowContract, partyArbitrator,
escrowId, false)
            ).to.be.revertedWith('ERC20: transfer amount exceeds
balance');
        });

        it('Sending ERC20 fees to arbitrator', async () => {
            await createEscrow(
              escrowContract,
              partyA,
              partyB.address,
              partyArbitrator.address,
              arbitratorFeeBps,
              'test',
              resolveTime,
              token.address,
              numToEth(amount),
              feeToken.address,
              numToEth(amount)
            );
            await depositFunds(escrowContract, partyB, escrowId);
            await time.increase(3601);
            await expect(
                determineOutcome(escrowContract, partyArbitrator,
escrowId, false)
            ).to.be.revertedWith('ERC20: transfer amount exceeds
balance');
            partyArbitratorToken1Balance = await
getTokenBalance(token, partyArbitrator.address);
            partyArbitratorToken2Balance = await
getTokenBalance(feeToken, partyArbitrator.address);

expect(ethToNum(partyArbitratorToken1Balance)).to.equal(0);

expect(ethToNum(partyArbitratorToken2Balance)).to.equal(0);
        });
    });
```

Running `npx hardhat test test/unit/TagEscrow.determineOutcome.FakeFeeToken.test.js` results in:

```
  TagEscrow.determineOutcome()
    Reverts
      ✓ When determining for partyA
      ✓ Sending ERC20 fees to arbitrator
```

This shows that if at least one of the ERC20s used to fund the escrow have a fee on transfer, calling `determineOutcome` will result in an `ERC20: transfer amount exceeds balance` revert. Note that this just tests for `determineOutcome`, however it will be a problem in any function that attempts to send funds to a party.

Recommendations

Using `SafeERC20` provides an additional layer of safety when interacting with ERC20 tokens, but it does not guarantee complete compliance with the ERC20 standard. Consider maintaining a whitelist of compliant ERC20 tokens to interact with the protocol. This can help ensure that only well-vetted and ERC20-compliant tokens are used. A resource like tokenlists.org can be used as a starting point (https://tokenlists.org/).

Otherwise, consider checking the contract's balance before and after a transfer to calculate the actual amount received.

Team response

"It's up to the users to select correctly functioning tokens and we offer a preselected list through our UI."

[H-03] Incentive for arbitrator to misbehave

Severity

Impact: High, because a winner can be falsely named as a loser and have their original funds and winnings stolen.

Likelihood: High, because there is currently limited incentive for arbitrators to behave.

Description

If the arbitrator doesn't make a decision, users are returned their amounts. Therefore, there is a strong incentive for the "losing" party to communicate with the arbitrator externally and pay them to not finalise the arbitration.

Recommendations

(1) Arbitrator stake
The arbitrator should provide collateral that is a % of party A + party Bs total tokens in the escrow. If they do not arbitrate then this is slashed. This gives an opportunity for another 3rd party arbitrator to provide a result and gain the slashed stake, if both parties agree.

(2) Arbitrator reputation system
Consider building a reputation system based on a history of results of all arbitrators so parties can view arbitrator history before agreeing to work with them.

Team response

"We are aware of this and currently in the process of building a reputation system. We believe that this should be informational vulnerability as it is behavioural and can be mitigated by social aspects."

[M-01] Lack of event emission after sensitive actions

Description

The `updateNftContractAddress` and `updateTreasuryAddress` functions of the `TagEscrow` contract do not emit relevant events after executing sensitive actions of updating addresses.

Recommendations

Consider emitting events, and potentially a timelock, after sensitive changes take place, to facilitate tracking and notify off-chain clients following the contract's activity.

[M-02] Lack of validation when setting `determineTime` and respective escape hatch

Description

It checks that the bet should be after the current time. However, due to the fact that `reclaimFunds` is the only escape hatch for an unresponsive arbitrator, there should be either some limit on the `determineTime` or another way of releasing the funds.

A problematic scenario is when Party A intentionally or mistakenly sets a `determineTime` years away and the arbitrator isn't responsive. If so, both parties will have to wait years to recover funds.

Recommendations

Consider requiring a maximum `determineTime` and consider implementing another function that, if called by both parties, allows `reclaimFunds` to be called.

Team response

"This is by design and is clear for users. Some events might be years away and we purposefully want to allow users the flexibility to decide."

[M-03] Not following checks-effects-interactions pattern

Description

This finding highlights a dangerous pattern used throughout the codebase that may eventually lead to exploitable scenarios if it continues to be followed.

Recommendations

Consider following the checks-effects-interactions pattern, using a mutex like the OpenZeppelin `nonReentrant` modifier may not offer enough protection from potential cross-function or cross-contract reentrancy attacks from logic introduced at a later stage.

[L-01] Unused Event Log

Description

`EscrowStarted` log event is declared but never emitted.

Recommendations

Ensure that the event is emitted at the appropriate point to provide visibility into contract activity.

[L-02] Lack of indexed parameters in events

Description

None of the parameters in the events defined in the contract are indexed.

Recommendations

Consider indexing event parameters to avoid hindering the task of off-chain services searching and filtering for specific events. This is especially relevant if you intend to leverage off-chain services to build reputation systems.

[L-03] Named return variables

Description

There is an inconsistent use of named return variables across the codebase.

Recommendations

For readability, either add a return statement of `escrowsParticipated` in `getEscrowsForAddress` or remove named return variables from all getters to maintain consistency.

[L-04] Missing error message in `require` statement

Description

The `require` statement in the `updateNftContractAddress` function is missing an error message.

Recommendations

Consider including a specific and informative error message in the `require` statement to provide better feedback on the cause of failure if the condition is not met.

[L-05] `ArbitratorFeeBps = 0` (unnecessary gas)

Description

Is there a time when you want to set this fee to zero? If not, require a minimum greater than 0 to save on gas.

Recommendations

Evaluate whether it is necessary to allow the `ArbitratorFeeBps` to be set to zero. If not, require a minimum greater than 0. If the fee can be set to zero and the deposited currency is ether, optimise the code to avoid unnecessary gas costs by not making a call to `tagFeeVault` when the fee is zero.

[I-01] Improve inline documentation for getter functions

Description

Without comments, it is harder to reason about the getter methods and how they are supposed to be used.

Recommendations

Consider improving the inline documentation for getter functions to provide better context on their intended use and behaviour. This will make it easier for other developers to understand the code and contribute to the project.

[I-02] Unspecific compiler version pragma

Description

The contract uses an unspecific compiler version pragma, which may lead to unintended behaviour due to compiler differences.

Recommendations

Avoid floating pragmas and instead use a specific compiler version pragma to ensure consistent behaviour across different compiler versions.

[I-03] Manual setting of NFT contract address

Description

The NFT contract address must be manually set, increasing the risk of misconfiguration.

Recommendations

Consider automating the process of setting the NFT contract address in the deploy script or implementing checks to ensure that the address is set correctly before the contract is used.

[I-04] Use of uint instead of uint256

Description

There are some instances of `uint`, as opposed to `uint256`, in the code.

Recommendations

In favour of explicitness, consider replacing instances of `uint` with `uint256` to ensure consistent usage of variable types and reduce potential confusion.