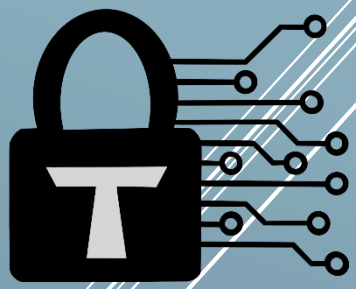
Trust Security



Smart Contract Audit

Abstract Arcade

15/11/2024

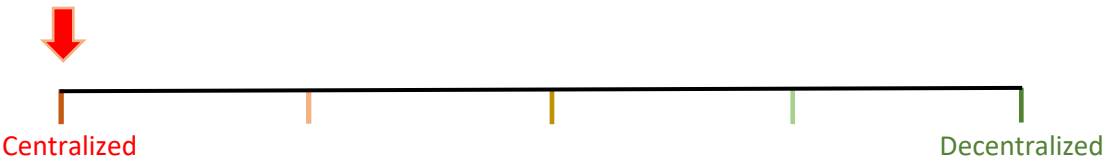
Executive summary

Category	GameFi
Audited file count	4
Lines of Code	277
Auditor	HollaDieWaldfee, ether_sky
Time period	11/11/2024 - 15/11/2024

Findings

Severity	Total	Open	Fixed	Acknowledged
High	1	1	0	0
Medium	3	3	0	0
Low	6	6	0	0

Centralization score



Signature

Digitally signed by Trust
Date: 2024.11.15 11:32:40 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	15/11/2024	Client report

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- src/Arcade.sol
- src/GiveawayPolicy.sol
- src/MulRewardPolicy.sol
- src/Multicall4.sol

Repository details

- **Repository URL:** https://github.com/maketh-labs/arcade
- **Commit hash:** 54ea90a2902c7b754d2ce68481f97719495e754d

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

HollaDieWaldfee is a renowned security expert with a track record of multiple first places in competitive audits. He is a Lead Auditor for Trust Security and Renascence Labs and Lead Senior Watson at Sherlock.

Ether_sky is a security researcher with a focus on blockchain security. He specializes in algorithms and data structures and has a solid background in IT development. He has placed at the top of audit contests in Code4rena and Sherlock recently.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	Good	Code is mostly simple and well structured.
Documentation	Moderate	Project could improve documentation.
Best practices	Good	Project adheres to best practices.
Centralization risks	Severe	Protocol owner has full custody of deposited funds.

Findings

High severity findings

TRST-H-1: Payout signature must include both the game solver and the puzzle ID

- **Category:** Signature issues
- **Source:** Arcade.sol
- **Status:** Open

Description

When a player solves a puzzle, the **answer** address signs the payout data, allowing the player to claim escrowed funds. However, the current payout signature lacks the game solver's address and the puzzle ID, leading to two issues.

Firstly, if the solver (e.g., User A) solves the puzzle but cannot claim their rewards (by calling the *solve()* function) before the game's expiry, others can exploit this.

Once the game expires, anyone can reset the game by calling the *expire()* function, and then start a new round with the *coin() function*.

```
function expire(Puzzle calldata puzzle) external payable returns (bool success) {
    if (plays < puzzle.lives) {
        statusOf[puzzleId] = uint256(plays) << 64;
    } else {
        statusOf[puzzleId] = INVALIDATED;
    }
    // Unfreeze assets.
    uint256 escrow = escrowOf[puzzleId];
    lockedBalanceOf[puzzle.currency][puzzle.creator] -= escrow;
    availableBalanceOf[puzzle.currency][puzzle.creator] += escrow;
    emit Expire(puzzleId);
    return true;
}
```

Suppose User A solves the puzzle, but by the time they call *solve()*, the puzzle has expired and the transaction fails. Using the payout signature for User A, the next player can immediately claim the funds by calling *coin()* and *solve()*, even within a single transaction via *multicall()*.

This prevents the rightful solver from receiving their rewards, while at the same time giving the rewards to a player that has not solved the puzzle.

For the second issue, consider two puzzles, P1 and P2, that share the same **answer**. A payout signature for P1 can also be used for P2. This means a player of P2 can claim rewards for solving P2 using P1's payout signature, even if P2 wasn't solved.

Recommended mitigation

It is recommended to use an EIP712 signature that includes the **payoutData**, **puzzle ID**, and the **game solver**. The reason for using an EIP712 signature is that it provides a domain separator such that multiple Arcade deployments can be active at the same time on any number of chains without users having to care about signature collisions.

What is not mitigated by this approach is that a game can expire before a user has had the chance to call *solve()* even when they did solve the game in time. However, this can be considered acceptable since technically *solve()* has not been called in time. It is just a matter of defining when a game counts as solved.

Team response

TBD.

Medium severity findings

TRST-M-1: EIP-6492 is not supported due to declaring *isValidSig()* as view

- **Category:** Standardization issues
- **Source:** IVerifySig.sol
- **Status:** Open

Description

UniversalSigValidator.isValidSig() implements EIP-6492 which performs an external *call()* to deploy the signer's account. Due to declaring the function as **view** in the interface, the function call to *isValidSig()* is compiled to a staticcall which does not allow for state changes to occur. Therefore, EIP-6492 is not supported.

Recommended mitigation

IVerifySig.isValidSig() must not be declared **view**. Note that even though callbacks to a user-controlled address are now possible, this will not allow for any reentrancy attacks.

```
interface IVerifySig {
-    function isValidSig(address _signer, bytes32 _hash, bytes memory _signature)
external view returns (bool);
+    function isValidSig(address _signer, bytes32 _hash, bytes memory _signature)
external returns (bool);
 }
```

Team response

TBD.

TRST-M-2: Malicious users can block others from playing a game

- **Category:** Griefing issues
- **Source:** Arcade.sol
- **Status:** Open

Description

In games that use the *GiveawayPolicy*, players are not required to pay a **toll**, allowing anyone to play with zero investment.

```
function escrow(uint256 toll, bytes calldata rewardData) external pure returns
(uint256) {
    if (toll != 0) {
        revert("GiveawayPolicy: Toll must be zero");
    }
    return abi.decode(rewardData, (uint256));
}
```

Therefore, malicious users can front-run the *coin()* function, preventing other players from participating in the game while at the same time temporarily locking the funds of the puzzle creator in **escrow**.

Also, if the game has a finite number of **lives**, malicious users can repeatedly call the *coin()* and *expire()* functions. This quickly exhausts the game's available **plays**, rendering it invalid and preventing anyone else from playing.

```
function expire(Puzzle calldata puzzle) external payable returns (bool success) {
    assembly {
        player := shr(96, status)
        plays := add(shr(64, status), 1)
    }
    if (plays < puzzle.lives) {
        statusOf[puzzleId] = uint256(plays) << 64;
    } else {
        statusOf[puzzleId] = INVALIDATED;
    }
}
```

Recommended mitigation

Allow restricting games to specific players or require a minimum **toll**. The incentives of requiring a larger **toll** to mitigate griefing attempts are described in TRST-SR-2. If the game is open to anyone, it is not possible to fully eliminate the issue, only the incentives can be set up to make griefing less likely. In scenarios where **toll** is intended to be zero, a whitelist of trusted players can be used instead.

Team response

TBD.

TRST-M-3: Expected rewards change if the payoutFee or creatorFee are adjusted while the game is in progress

- **Category:** Time-sensitivity issues
- **Source:** Arcade.sol
- **Status:** Open

Description

Rational players and puzzle creators calculate expected rewards before deciding to join a game or create a game. The **payoutFee** configuration value directly impacts the final rewards determined in the *solve()* function, and a change in the **payoutFee** by the protocol owner can make a game, that has initially been determined profitable (positive expected reward), unprofitable (negative expected reward).

```
function solve(Puzzle calldata puzzle, bytes32 payoutData, bytes calldata
payoutSignature) external {
    // Settle reward.
    uint256 escrow = escrowOf[puzzleId];
    uint256 payout = IRewardPolicy(puzzle.rewardPolicy).payout(escrow, payoutData);
    if (escrow < payout) {
        revert("Arcade: Payout is greater than escrow");
    }
    uint256 protocolFee = payout * payoutFee / FEE_PRECISION;
    lockedBalanceOf[puzzle.currency][puzzle.creator] -= escrow;
    availableBalanceOf[puzzle.currency][owner()] += protocolFee;
    availableBalanceOf[puzzle.currency][puzzle.creator] += escrow - payout;
    availableBalanceOf[puzzle.currency][player] += payout - protocolFee;
    emit Solve(puzzleId, payout);
```

```
}
```

Similarly, for a puzzle creator, a change in the **creatorFee** can make a puzzle unprofitable without being able to invalidate it.

Recommended mitigation

Each time a player calls *coin()* to play a game, **payoutFee** should be snapshotted, such that a later change in the **payoutFee** does not impact the user's payout. It is also necessary that players can supply the expected **payoutFee** to *coin()* and to revert execution if the expected fee does not match the actual fee. Both mitigations together allow a player to make an expected payout calculation before joining a game and be sure the calculation can't be affected by an action that the protocol owner performs.

Additionally, each puzzle should contain an expected **creatorFee** which is checked against the actual **creatorFee** at the first time a player plays the game. At this time the **creatorFee** should be snapshotted and then be applied to all subsequent plays.

Team response

TBD.

Low severity findings

TRST-L-1: Prevent puzzles with zero lives

- **Category:** Validation issues
- **Source:** Arcade.sol
- **Status:** Open

Description

A puzzle with zero lives should not be playable. However, the lives are only checked in *expire()*, and so a puzzle with zero lives can still be played and one attempt can be made to solve it. Since creating a puzzle with zero lives is not a reasonable use case, the finding is only of Low severity.

Recommended mitigation

It should be validated in *coin()* that a puzzle does not have zero lives.

```
@@ -121,6 +122,10 @@ contract Arcade is IArcade, Ownable2Step, Multicall4, EIP712 {
             revert("Arcade: Puzzle deadline exceeded");
         }

+        if (puzzle.lives == 0) {
+            revert("Arcade: Invalid lives");
+        }
```

Team response

TBD.

TRST-L-2: Users lack incentive to call *expire()*

- **Category:** Game-theory flaws
- **Source:** Arcade.sol
- **Status:** Open

Description

Puzzles that are acquired by a user can be released by calling *expire()*. To call the function, either the expiration time must be reached, or the caller must be the player that has acquired the puzzle.

Players don't have any incentive to call the function before the expiration is reached since there is the downside of no longer being able to solve the puzzle and paying the transaction fee, while at the same time there is no upside.

Recommended mitigation

Puzzles can eventually be expired by anyone, and it is not clear that an incentive for premature expiration is a mechanic that is intended. If it is determined that such an incentive is missing, a possible solution is for the **toll** to only be processed in *expire()* and *solve()* and to refund part

of the **toll**, for example, decreasing linearly with the time that the puzzle has been locked, to the player upon calling *expire()*.

Team response

TBD.

TRST-L-3: Puzzle can be expired but also solvable

- **Category:** Logical issues
- **Source:** Arcade.sol
- **Status:** Open

Description

When **block.timestamp = expiryTimestamp**, it is possible to call both *solve()* and *expire()*. This is inconsistent since either the puzzle is expired and it can't be solved anymore, or it can be solved and hasn't expired yet. Beyond the logical inconsistency there isn't any impact, however the griefing impact of frontrunning a valid *solve()* with *expire()* should be stated.

Recommended mitigation

The **expiryTimestamp**, by definition, is the timestamp at which a puzzle should expire. So, the check in *solve()* needs to be more restrictive to not allow solving an expired puzzle.

```
@@ -201,7 +209,7 @@ contract Arcade is IArcade, Ownable2Step, Multicall4, EIP712 {
         }

         // Make sure game hasn't expired.
-        if (uint64(block.timestamp) > uint64(status)) {
+        if (uint64(block.timestamp) >= uint64(status)) {
             revert("Arcade: Puzzle has expired");
         }
```

Team response

TBD.

TRST-L-4: Signature validation is missing zero address check

- **Category:** Validation issues
- **Source:** Arcade.sol
- **Status:** Open

Description

UniversalSigValidator.isValidSig() returns **true** if **_signer = address(0)** and an invalid signature is supplied. This can be abused to forge any puzzle with **address(0)** as the creator and to steal the funds of **address(0)** if it holds any. Also, if a puzzle has set **answer = address(0)**, it can be solved by providing an invalid signature.

Since both cases do not reflect intended usage and don't impact other puzzles, the finding is of Low severity.

Recommended mitigation

In *coin()*, it must be checked that neither **creator**, nor **answer** are set to **address(0)**.

```
@@ -121,6 +122,14 @@ contract Arcade is IArcade, Ownable2Step, Multicall4, EIP712 {
             revert("Arcade: Puzzle deadline exceeded");
         }

+        if (puzzle.creator == address(0)) {
+            revert("Arcade: Invalid creator");
+        }
+
+        if (puzzle.answer == address(0)) {
+            revert("Arcade: Invalid answer");
+        }
```

Team response

TBD.

TRST-L-5: Using *transferFrom()* instead of *safeTransferFrom()* leads to token incompatibility

- **Category:** Logical issues
- **Source:** Arcade.sol
- **Status:** Open

Description

Arcade uses *SafeERC20* for all token transfers, except for one instance in *coin()* where *transferFrom()* is used. This makes the protocol incompatible with tokens that don't return a bool (e.g., USDT). For now, only USDC and WETH will be used with the protocol, and there is no issue with these tokens.

However, to be able to use all common tokens in the future, and to be consistent, *SafeERC20* should be used in all instances.

Recommended mitigation

Consistently use *safeTransferFrom()* instead of *transferFrom()*.

```
            uint256 available = availableBalanceOf[currency][msg.sender];
            if (toll > available) {
-                IERC20(currency).transferFrom(msg.sender, address(this), toll -
available);
+                IERC20(currency).safeTransferFrom(msg.sender, address(this), toll -
available);
                 availableBalanceOf[currency][msg.sender] = 0;
            } else {
                 availableBalanceOf[currency][msg.sender] -= toll;
```

Team response

TBD.

TRST-L-6: New reward policies can introduce reentrancy vulnerabilities

- **Category:** Reentrancy issues
- **Source:** Arcade.sol
- **Status:** Open

Description

While the protocol currently only supports two reward policies, *GiveawayPolicy* and *MulRewardPolicy*, it is possible to introduce new reward policies that also make external calls. If such a reward policy is introduced, the call to *IRewardPolicy.escrow()* in *coin()* can be exploited by calling *coin()* again. By the second time that *coin()* is called, **statusOf[puzzleId]** still has the old value, and so the **escrow** is locked a second time. Since the game is only created once, the additional escrowed funds are lost.

Recommended mitigation

Inherit from OZ's *ReentrancyGuard* and apply the **nonReentrant** modifier to all state-changing functions from which a callback can potentially be initiated, this includes ERC20 transfers to provide security for ERC777 tokens. In the *coin()* function, the **nonReentrant** modifier must be placed before the **validatePuzzle** modifier so that signature validation is performed with the reentrancy guard enabled.

This approach of using a reentrancy guard provides enhanced security without having to reason about individual reentrancy scenarios.

Team response

TBD.

Additional recommendations

TRST-R-1: Cap fees below a hundred percent

It is recommended to limit **creatorFee** and **payoutFee** to a reasonable value like 10% to reduce centralization risks.

```
     function setCreatorFee(uint256 _newFee) external onlyOwner {
-        if (_newFee > FEE_PRECISION) {
-            revert("Arcade: Fee cannot be greater than or equal to 100%");
+        if (_newFee > FEE_PRECISION / 10) {
+            revert("Arcade: Fee cannot be greater than or equal to 10%");
         }
         uint256 oldFee = creatorFee;
         creatorFee = _newFee;
@@ -270,8 +270,8 @@ contract Arcade is IArcade, Ownable2Step, Multicall4, EIP712 {
     }

     function setPayoutFee(uint256 _newFee) external onlyOwner {
-        if (_newFee > FEE_PRECISION) {
-            revert("Arcade: Fee cannot be greater than or equal to 100%");
+        if (_newFee > FEE_PRECISION / 10) {
+            revert("Arcade: Fee cannot be greater than or equal to 10%");
         }
         uint256 oldFee = payoutFee;
         payoutFee = _newFee;
```

TRST-R-2: Remove payable modifier from *expire()*

Since *expire()* does not contain logic that uses ETH, the **payable** modifier should be removed to avoid mistakes by users.

```
-    function expire(Puzzle calldata puzzle) external payable returns (bool success) {
+    function expire(Puzzle calldata puzzle) external returns (bool success) {
         bytes32 puzzleId = keccak256(abi.encode(puzzle));
         uint256 status = statusOf[puzzleId];
         address player;
```

TRST-R-3: Allow the *coin()* function to directly accept ETH deposits

Currently, users can deposit ETH through the *depositETH()* function, but the *coin()* function does not support accepting ETH. As a result, users must call *depositETH()* separately before calling the *coin()* function if the **currency** is WETH. Adding direct ETH acceptance to the *coin()* function will streamline the process for users.

TRST-R-4: Add a feature to refund excess ETH in the *depositETH()* function

Currently, when users deposit ETH through *depositETH()*, excess ETH is not refunded. It should be considered to implement a refund functionality. However, the straightforward implementation where **msg.value – amount** is refunded does not work as expected when

multiple ETH deposits are made within a single *multicall()* as the first ETH deposit will refund the ETH provided for a later ETH deposit. The suggested solution checks if *depositETH()* is called from within a *multicall()*, and if so, does not perform a refund. The refund is then only performed within *multicall()*. The fix must also encompass *coin()* if *coin()* is changed to allow ETH deposits as suggested in TRST-R-3 above.

Note the **require(!isMulticall)** in *multicall()* which acts as a reentrancy guard to prevent an inner *multicall()* from refunding the ETH of an outer *multicall()*.

```
diff --git a/src/Arcade.sol b/src/Arcade.sol
index 21a792f..1067b55 100644
--- a/src/Arcade.sol
+++ b/src/Arcade.sol
@@ -75,6 +75,10 @@ contract Arcade is IArcade, Ownable2Step, Multicall4, EIP712 {
     function depositETH(address user, uint256 amount) external payable {
         IWETH(WETH).deposit{value: amount}();
         availableBalanceOf[WETH][user] += amount;
+        if (!isMulticall && address(this).balance > 0) {
+            (bool success, ) = msg.sender.call{value: address(this).balance}("");
+            require(success, "Arcade: ETH refund failed");
+        }
         emit Deposit(user, WETH, amount);
     }

diff --git a/src/Multicall4.sol b/src/Multicall4.sol
index 3b716c8..9c08ead 100644
--- a/src/Multicall4.sol
+++ b/src/Multicall4.sol
@@ -6,7 +6,10 @@ import {IMulticall4} from "./interfaces/IMulticall4.sol";
 /// @title Multicall4
 /// @notice Enables calling multiple methods in a single call to the contract
 abstract contract Multicall4 is IMulticall4 {
+    bool internal isMulticall;
     function multicall(bytes[] calldata data) external payable returns (bytes[]
memory results) {
+        require(!isMulticall, "Arcade: reentered multicall");
+        isMulticall = true;
         results = new bytes[](data.length);
         for (uint256 i = 0; i < data.length; i++) {
             (bool success, bytes memory result) =
address(this).delegatecall(data[i]);
@@ -20,5 +23,10 @@ abstract contract Multicall4 is IMulticall4 {

             results[i] = result;
         }
+        if (address(this).balance > 0) {
+            (bool success, ) = msg.sender.call{value: address(this).balance}("");
+            require(success, "Arcade: ETH refund failed");
+        }
+        isMulticall = false;
     }
 }
```

TRST-R-5: Deadline does not include the timeLimit for solving the puzzle

While the **Puzzle** struct includes a **deadline** field, which logically implies the game should end by that time, users can still continue playing even after the **deadline** since it only restricts the time at which *coin()* can be called.

It is not clear if **deadline** should define the boundary at which a user can no longer start playing the game, or at which the game should not be solvable anymore.

In the former case, the code works correctly. In the latter case, the code can be updated as follows:

```
function coin(Puzzle calldata puzzle, bytes calldata signature, uint256 toll)
    external
    payable
    validatePuzzle(puzzle, signature)
{
    uint64 expiryTimestamp = uint64(block.timestamp) + puzzle.timeLimit;

+    if (expiryTimestamp > puzzle.deadline) {
+        expiryTimestamp = puzzle.deadline;
+    }

    {
        assembly {
            status := add(shl(96, player), add(shl(64, plays), expiryTimestamp))
        }
        statusOf[puzzleId] = status;
    }
 }
```

Centralization risks

TRST-CR-1: Protocol owner is allowed to set arbitrary fees

The privileged functions that the protocol owner is allowed to call are *setCreatorFee()* and *setPayoutFee()*. Creator fees are charged when a game is played in *coin()*, and the fees are a percentage of the **toll** that a user pays to the puzzle creator. Payout fees are charged in *solve()* and subtracted from the **payout** that a user receives.

Both fees can currently be set as high as 100%, and a change in the payout or creator fee is not limited to new games. Instead, it affects ongoing games. It is recommended to reduce centralization risks by introducing a maximum fee and to apply payout and creator fee changes only to new games as described in TRST-M-3.

TRST-CR-2: Creators are trusted for their own puzzles

Puzzle creators have full control over the parameters that they create puzzles with. For discussing centralization risks, the relevant parameters are **answer**, i.e., the address that needs to sign the puzzle solution, **currency** and **rewardPolicy**, which are external contracts that the protocol interacts with.

The **answer** address is trusted to sign the solution for the correct user. If **answer** becomes inactive or signs a solution for the wrong user, the legitimate user does not receive their payout.

Users must verify that **currency** and **rewardPolicy** are legitimate and contain the intended logic. Here, the worst possible impact for the user is a loss of the **toll** without being able to solve the game and a loss of the potential **payout**. It must be noted that for the creator itself, the consequences of providing a wrong **currency** and **rewardPolicy** can be worse, since due to reentrancy, the creator's funds can be locked. The consequences of reentrancy are discussed in TRST-L-6 and security measures are suggested.

Finally, creators are able to call *invalidate()* and prevent users from playing games for which they have a valid signature from the creator but that nobody has played yet. The same outcome can be achieved by the creator by withdrawing all available balance from their account.

Systemic risks

TRST-SR-1: Protocol can be used with any token and reward policy

Fundamentally, the protocol can be used with any ERC20 token and reward policy. The protocol has been audited based on the expected tokens (USDC and WETH) and reward policies (*GiveawayPolicy* and *MulRewardPolicy*).

Using different tokens or reward policy implementations can lead to a loss of funds for both players and creators. It has been validated that one puzzle created with bad parameters should not have an impact on other puzzles.

TRST-SR-2: Protection against griefing attacks is probabilistic and depends on puzzle configuration

Anyone is able call *coin()* and acquire the puzzle. This means other users are not able to interact with the puzzle and the funds of the creator are locked in escrow.

The incentive to perform this griefing attack is positively correlated with the **timeLimit** (how long one griefing attack lasts) and **escrow** (how much funds can be locked with one griefing attack). Conversely, there is a negative correlation with the **toll** that an attacker needs to pay to acquire the puzzle.

For example, in one scenario, 100 WETH can be locked for 1 year by paying a toll of 0.1 WETH. In a second scenario, 1 WETH can be locked for 1 day by paying a toll of 0.9 WETH. The first configuration has a high griefing risk, while the second does not.

Assessing the risk quantitatively has not been part of the audit.