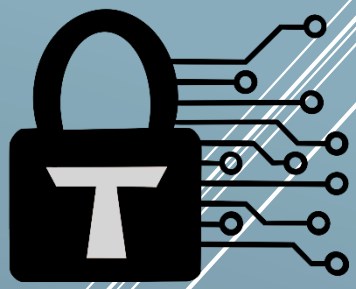
Trust Security



Smart Contract Audit

MakeETH - Floor

06/08/2025

Executive summary

Category	Gaming
Audited file count	2
Lines of Code	476
Auditor	HollaDieWaldfee
Time period	04/08/2025 - 06/08/2025

Findings

Severity	Total	Open	Fixed	Acknowledged
High	2	2	0	0
Medium	1	1	0	0
Low	1	1	0	0

Centralization score

Signature

Digitally signed by Trust
Date: 2025.08.06 11:55:12 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	06/08/2025	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/CommitReveal.sol
- src/SignedVault.sol

Repository details

- **Repository URL:** https://github.com/maketh-labs/floor
- **Commit hash:** c893277ed3098c4e0ce84edd76dd19287d23c4c8

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

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	**Excellent**	Project kept code as simple as possible, reducing attack risks.
Documentation	**Good**	Project has inline comments and basic documentation in README.
Best practices	**Good**	Project adheres to best practices.
Centralization risks	**Severe**	Protocol owner is fully trusted. Resolvers have limited trust assumptions.

Findings

High severity findings

TRST-H-1: Reentrancy in deposit functions enables manipulation of deposits mapping

- **Category:** Reentrancy attacks
- **Source:** SignedVault.sol
- **Status:** Open

Description

The *deposit()* and *depositWithPermit2()* functions can be called with any **token**, thus allowing an attacker to receive a callback.

```
    function deposit(address token, uint256 amount, address resolver, uint256 nonce)
external {
        if (token == ETH_ADDRESS) revert InvalidAsset();
        if (resolver == address(0)) revert InvalidResolver();


        // Check for duplicate deposits
        if (deposits[msg.sender][nonce] != 0) revert DuplicateDeposit(msg.sender,
nonce);


        IERC20(token).safeTransferFrom(msg.sender, address(this), amount);


        // Store deposit for backend verification
        deposits[msg.sender][nonce] = amount;


        resolverBalanceOf[resolver][token] += amount;
        emit Deposit(msg.sender, token, amount, nonce);
    }
```

When the attacker receives a callback, **deposits[msg.sender][nonce]** has not yet been updated, and a second **Deposit** event with the same **nonce** can be emitted by reentering either of the two functions. Furthermore, the final value that persists in the mapping is the value from the outer call.

If the outer call is made with a malicious token that has an arbitrary supply and grants the attacker a callback, the inner call can be used to emit an event with a legitimate token like USDC and a 1 wei **amount**. The backend can thus be tricked into reading the arbitrary **amount** from the outer call and to connect it with the USDC **Deposit** event. In terms of the business logic, this means an attacker can play a game without making a deposit.

Recommended mitigation

Consistently apply reentrancy guards to all functions in *SignedVault* (including *cancelSignature()* for best practice).

```
diff --git a/src/SignedVault.sol b/src/SignedVault.sol
index fa5a37e..4bb43eb 100644
```

```
--- a/src/SignedVault.sol
+++ b/src/SignedVault.sol
@@ -114,7 +114,7 @@ contract SignedVault is
      * @param resolver Address of the resolver who will be responsible for this
deposit
      * @param nonce User-provided nonce for deposit verification
      */
-    function depositETH(address resolver, uint256 nonce) external payable {
+    function depositETH(address resolver, uint256 nonce) external payable
nonReentrant {
         if (resolver == address(0)) revert InvalidResolver();

         // Check for duplicate deposits
@@ -137,7 +137,7 @@ contract SignedVault is
      * @param resolver Address of the resolver who will be responsible for this
deposit
      * @param nonce User-provided nonce for deposit verification
      */
-    function deposit(address token, uint256 amount, address resolver, uint256 nonce)
external {
+    function deposit(address token, uint256 amount, address resolver, uint256 nonce)
external nonReentrant {
         if (token == ETH_ADDRESS) revert InvalidAsset();
         if (resolver == address(0)) revert InvalidResolver();

@@ -165,7 +165,7 @@ contract SignedVault is
         ISignatureTransfer.PermitTransferFrom memory permit,
         bytes calldata signature,
         uint256 nonce
-    ) external {
+    ) external nonReentrant {
         address token = permit.permitted.token;
         uint256 amount = permit.permitted.amount;

@@ -203,7 +203,7 @@ contract SignedVault is
      * @param signature The signature to cancel
      */
     function cancelSignature(address user, address token, uint256 amount, uint256
deadline, bytes calldata signature)
-        external
+        external nonReentrant
     {
         // Check if signature has already been used or cancelled
         bytes32 signatureHash = keccak256(signature);
```

Team response

TBD.

TRST-H-2: Deposits with zero amount allow to emit multiple Deposit events with the same nonce and manipulate deposits mapping

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

Description

SignedVault protects against multiple deposits with the same **nonce** by requiring that **deposits[msg.sender][nonce] == 0**. However, the deposit functions can be called with

amount=0 or **msg.value=0**. Consequently, multiple **Deposit** events can be emitted with the same **nonce**. The last deposit can then be performed with **amount > 0** to trick the backend.

Similar to finding TRST-H-1, when the backend processes the **Deposit** events and reads the **amount** stored in the **deposits** mapping, the state is inconsistent, potentially allowing users to participate in games for free.

For example, the following two events may be observed:

- **Deposit(user,USDC,0,1)**
- **Deposit(user,tokenXYZ,100e18,1)**

Due to the second deposit for **tokenXYZ**, the **deposits** mapping for **nonce=1** shows an amount of **1e18**.

Recommended mitigation

Implement checks in *deposit()*, *depositETH()* and *depositWithPermit2()* to prevent deposits with zero **amount** or **msg.value**.

Team response

TBD.

Medium severity findings

TRST-M-1: Resolver must reveal signature when calling *cancelSignature()*

- **Category:** Front-running issues
- **Source:** SignedVault.sol
- **Status:** Open

Description

To cancel a signature in *SignedVault*, the signature must be made known on-chain since **usedSignatures** tracks signature hashes. Therefore, when a resolver calls *cancelSignature()*, the **signature** parameter can be extracted by anyone to then call *withdraw()* / *withdrawETH()* with it. If the signature is for a message that withdraws funds to an address that is different from the resolver itself, and the resolver has a sufficient balance, this can lead to a loss of funds.

To prevent front-running, the resolver can first withdraw all their balance and then call *cancelSignature()*. However, this flow is not intended, and it should be possible to just call *cancelSignature()* without any risk. The impact of the issue is somewhat mitigated by the consideration that for a signature that has not become public yet, the resolver does not need to cancel it, and a public signature can't be leaked in the first place. There remain edge cases where, e.g., the resolver suspects a signature could have become public.

Recommended mitigation

Signatures should contain a **nonce** such that signers can simply invalidate the **nonce** to cancel signatures. The **usedSignatures** mapping can then be deprecated in favor of the nonces. This has the additional benefit of being able to sign multiple withdrawals for the same user and amount.

In *CommitReveal*, the current signature mechanism is sufficient since it's not possible to cancel signatures, and there is no need to sign the same message twice given that **gameSeedHash** is a unique value for each game.

Team response

TBD.

Low severity findings

TRST-L-1: Game creations can be griefed due to missing front-running protection in signatures

- **Category:** Front-running issues
- **Source:** CommitReveal.sol
- **Status:** Open

Description

CommitReveal ensures that game creation signatures are only used once by maintaining a **usedSignatures** mapping. The **serverSignature** is signed by a **resolver** and its message contains the **msg.sender** which is the player.

Even if the **serverSignature** is submitted by a different **msg.sender** than it was intended for, there will be a valid signer address recovered. While it's not possible to forge the original **resolver**, the recovered address is a random address, and the **serverSignature** is marked as used. Thus, by front-running the user, their **serverSignature** can be invalidated. The issue provides a griefing vector where users may need to send multiple transactions until their **serverSignature** is accepted.

The issue can be verified with the following test. Note that **betAmount** is modified to **1 wei** to keep the cost of the attack as low as possible.

```
function testCreateGameETH() public {
        uint256 deadline = block.timestamp + 1 hours;

        CommitReveal.CreateGameParams memory params = CommitReveal.CreateGameParams({
            token: address(0), // ETH
            betAmount: BET_AMOUNT,
            gameSeedHash: GAME_SEED_HASH,
            algorithm: ALGORITHM,
            gameConfig: GAME_CONFIG,
            deadline: deadline
        });

        bytes memory signature = createGameSignature(params, player);
        console2.log("resolver: ", resolver);
        console2.log("player: ", player);

        // @audit-info We don't submit as the player and change betAmount to 1 wei to
make the attack cheap
        // vm.prank(player);
        params.betAmount = 1;
        commitReveal.createGame{value: 1}(params, signature, SALT);

        // Verify game was created correctly
        assertTrue(commitReveal.usedSignatures(keccak256(signature)));
    }
```

Recommended mitigation

The **resolver** address should be included in **CREATE_GAME_TYPEHASH** and **resolver == recoveredAddress** should be checked, such that the **serverSignature** becomes invalid when submitted by the wrong **player**. Finding a **resolver** for which the **(signature, message hash)** pair is valid is the usual case of guessing messages.

Team response

TBD.

Additional recommendations

TRST-R-1: Define __gap as the last storage variable

It is convention that **__gap** is defined as the last storage variable in a contract such that new variables are appended to, rather than inserted in between, existing storage variables.

```
diff --git a/src/CommitReveal.sol b/src/CommitReveal.sol
index 279326c..c0d21ce 100644
--- a/src/CommitReveal.sol
+++ b/src/CommitReveal.sol
@@ -77,9 +77,6 @@ contract CommitReveal is
     /// @notice Mapping of resolver balances by token
     mapping(address resolver => mapping(address token => uint256 balance)) public
balanceOf;

-    /// @notice Reserved slots for upgradeability
-    uint256[50] private __gap; // 50 reserved slots
-
     /// @notice Game status enum
     enum GameStatus {
         None, // Default state (0) - game doesn't exist
@@ -108,6 +105,9 @@ contract CommitReveal is
     /// @notice Mapping from game signature hash to Game data
     mapping(bytes32 signatureHash => Game game) internal _games;

+    /// @notice Reserved slots for upgradeability
+    uint256[50] private __gap; // 50 reserved slots
+
     /*´:°•.°+.*•´.*:˚.°*.˚•´.°:°•.°•.*•´.*:˚.°*.˚•´.°:°•.°+.*•´.*:*/
     /*                            EVENTS                            */
     /*.•°:°.´+˚.*°.˚:*.´•*.+°.•°:´*.´•*.•°.•°:°.´:•˚°.*°.˚:*.´+°.•*/
```

TRST-R-2: Remove unused / redundant code

- *CommitReveal* doesn't use **InvalidSignature** error.
- Checking the result of *ECDSA.recover()* for address zero is redundant and already done inside the library.

```
diff --git a/src/CommitReveal.sol b/src/CommitReveal.sol
index 279326c..81846e2 100644
--- a/src/CommitReveal.sol
+++ b/src/CommitReveal.sol
@@ -303,7 +303,7 @@ contract CommitReveal is
             bytes32 messageHash = _hashTypedDataV4(
                 keccak256(abi.encode(CASH_OUT_TYPEHASH, gameId, payoutAmount,
gameState, gameSeed, deadline))
             );
-            if (_verifyAndGetResolver(messageHash, serverSignature) !=
game.resolver) revert InvalidResolverSignature();
+            if (ECDSA.recover(messageHash, serverSignature) != game.resolver)
revert InvalidResolverSignature();
         }

         // Check resolver has sufficient balance
@@ -364,7 +364,7 @@ contract CommitReveal is
             bytes32 messageHash = _hashTypedDataV4(
                 keccak256(abi.encode(MARK_GAME_AS_LOST_TYPEHASH, gameId,
gameState, gameSeed, deadline))
```

```
            );
-           if (_verifyAndGetResolver(messageHash, serverSignature) !=
game.resolver) revert InvalidResolverSignature();
+           if (ECDSA.recover(messageHash, serverSignature) != game.resolver)
revert InvalidResolverSignature();
        }

        // Update game state
@@ -476,20 +476,6 @@ contract CommitReveal is
     /*                    INTERNAL FUNCTION                       */
     /*.•°:°.´+˚.*°.˚:*.´•*.+°.•°:´*.´•*.•°.•°:°.´:•˚°.*°.˚:*.´+°.•*/

-    /**
-     * @dev Verifies signature and returns the resolver address
-     * @param _hash The hash that was signed
-     * @param _signature The signature bytes
-     * @return resolver The address of the resolver who signed
-     */
-    function _verifyAndGetResolver(bytes32 _hash, bytes calldata _signature)
internal pure returns (address) {
-        address recoveredSigner = ECDSA.recover(_hash, _signature);
-        if (recoveredSigner == address(0)) {
-            revert InvalidResolverSignature();
-        }
-        return recoveredSigner;
-    }
-
     /**
      * @dev Verifies the signature of a game creation request and returns the
resolver address.
      * @param params Game creation parameters struct
@@ -516,7 +502,7 @@ contract CommitReveal is
                 )
             )
         );
-        return _verifyAndGetResolver(messageHash, serverSignature);
+        return ECDSA.recover(messageHash, serverSignature);
     }

     /**
diff --git a/src/SignedVault.sol b/src/SignedVault.sol
index fa5a37e..ec06818 100644
--- a/src/SignedVault.sol
+++ b/src/SignedVault.sol
@@ -217,7 +217,7 @@ contract SignedVault is

         // Verify that the caller is the resolver who signed this signature
         address recoveredSigner = ECDSA.recover(messageHash, signature);
-        if (recoveredSigner == address(0) || recoveredSigner != msg.sender) {
+        if (recoveredSigner != msg.sender) {
             revert InvalidSignature();
         }

@@ -309,7 +309,7 @@ contract SignedVault is
         }

         address recoveredSigner = ECDSA.recover(messageHash, signature);
-        if (recoveredSigner == address(0) || recoveredSigner != resolver) {
+        if (recoveredSigner != resolver) {
             revert InvalidSignature();
         }
```

- **transferDetails** parameter can be removed from *CommitReveal.createGameWithPermit2()* since the function has all necessary information to create the parameter by itself.

```
diff --git a/src/CommitReveal.sol b/src/CommitReveal.sol
index 279326c..e1a4c24 100644
--- a/src/CommitReveal.sol
+++ b/src/CommitReveal.sol
@@ -239,7 +239,6 @@ contract CommitReveal is
         bytes calldata serverSignature,
         bytes32 salt,
         ISignatureTransfer.PermitTransferFrom memory permit,
-        ISignatureTransfer.SignatureTransferDetails calldata transferDetails,
         bytes calldata permitSignature
     ) external nonReentrant {
         if (block.timestamp > params.deadline) revert SignatureExpired();
@@ -261,10 +260,9 @@ contract CommitReveal is
         // Verify permit amount is sufficient
         if (permit.permitted.amount < params.betAmount) revert
InsufficientPermitAmount();

-        // Verify transfer details are safe
-        if (transferDetails.to != address(this)) revert
InvalidPermitTransfer();
-        if (transferDetails.requestedAmount != params.betAmount) revert
InvalidAmount(transferDetails.requestedAmount);
-
+        ISignatureTransfer.SignatureTransferDetails memory transferDetails =
+            ISignatureTransfer.SignatureTransferDetails({to: address(this),
requestedAmount: params.betAmount});
+
         // Transfer tokens using Permit2
         PERMIT2.permitTransferFrom(permit, transferDetails, msg.sender,
permitSignature);
```

TRST-R-3: Call all initializers of parent contracts

It is best practice to call all initializers of parent contracts even if they are empty.

```
diff --git a/src/CommitReveal.sol b/src/CommitReveal.sol
index 279326c..df6ba47 100644
--- a/src/CommitReveal.sol
+++ b/src/CommitReveal.sol
@@ -173,8 +173,10 @@ contract CommitReveal is

     function initialize(address _owner) public initializer {
         __Ownable_init(_owner);
+        __Ownable2Step_init();
         __ReentrancyGuard_init();
         __EIP712_init("CommitReveal", "1");
+        __UUPSUpgradeable_init();
     }

     /*´:°•.°+.*•´.*:˚.°*.˚•´.°:°•.°•.*•´.*:˚.°*.˚•´.°:°•.°+.*•´.*:*/
diff --git a/src/SignedVault.sol b/src/SignedVault.sol
index fa5a37e..ccb7107 100644
--- a/src/SignedVault.sol
+++ b/src/SignedVault.sol
@@ -101,8 +101,10 @@ contract SignedVault is

     function initialize(address _owner) public initializer {
         __Ownable_init(_owner);
+        __Ownable2Step_init();
         __ReentrancyGuard_init();
         __EIP712_init("SignedVault", "1");
+        __UUPSUpgradeable_init();
     }
```

TRST-R-4: *CommitReveal.createGameWithPermit2()* should check that token is not ETH_ADDRESS

Since *CommitReveal.createGameWithPermit2()* does not support ETH, it is recommended to add an explicit check. Otherwise, the permit2 transfer fails by calling **address(0)**.

```
diff --git a/src/CommitReveal.sol b/src/CommitReveal.sol
index 279326c..ae1d7a5 100644
--- a/src/CommitReveal.sol
+++ b/src/CommitReveal.sol
@@ -244,6 +244,7 @@ contract CommitReveal is
     ) external nonReentrant {
         if (block.timestamp > params.deadline) revert SignatureExpired();
         if (params.betAmount == 0) revert InvalidAmount(params.betAmount);
+        if (params.token == ETH_ADDRESS) revert InvalidAsset();

         // Calculate game ID from signature hash
         bytes32 gameId = keccak256(serverSignature);
```

TRST-R-5: Add reentrancy guard to *CommitReveal.markGameAsLost()*

All functions in *CommitReveal* except for *markGameAsLost()* are protected with reentrancy guards. For consistency, *markGameAsLost()* should also be protected. The missing reentrancy guard does not have any impact since the only callbacks that could cause inconsistent state are from *createGame()* and *createGameWithPermit2()* and the game that is created does not have its status set to **Active** yet, preventing the call to *markGameAsLost()*.

Centralization risks

TRST-CR-1: Protocol owner is fully trusted

Both *CommitReveal* and *SignedVault* inherit from *UUPSUpgradeable* and are upgradeable by the **owner**. Thus, the **owner** can upgrade the proxies to arbitrary implementations and has full access to all funds that are owned by the contracts (including granted allowances and valid Permit2 signatures).

TRST-CR-2: Resolvers are trusted but isolated from each other

Resolvers are trusted to correctly manage their games. Malicious or inactive resolvers can decide not to resolve games or not deposit sufficient funds to pay for winning games. In *CommitReveal*, honest behavior by resolvers can be verified but not enforced. Importantly, resolvers are isolated from each other. One resolver is not able to interfere with games that are managed by other resolvers.

Systemic risks

TRST-SR-1: External token risk

While the contracts are designed such that tokens can't affect each other, users can suffer a complete loss of funds if the token they choose to interact with behaves maliciously or unexpectedly or simply loses its value. Due to the use of *SafeERC20* for all token interactions, the contracts are compatible with most common ERC20 tokens. Native tokens are also supported.