# Outcome

SMART CONTRACTS REVIEW

zokyo

June 6th 2025 | v. 1.0

# Security Audit Score

## PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

# # ZOKYO AUDIT SCORING OUTCOME

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

# SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 2 Critical issues: 2 resolved = 0 points deducted
- 3 High issues: 3 resolved = 0 points deducted
- 6 Medium issues: 5 resolved and 1 acknowledged = - 3 points deducted
- 2 Low issues: 2 resolved = 0 points deducted
- 5 Informational issues: 2 resolved and 3 acknowledged = 0 points deducted

Thus, 100 - 3 = 97

# TECHNICAL SUMMARY

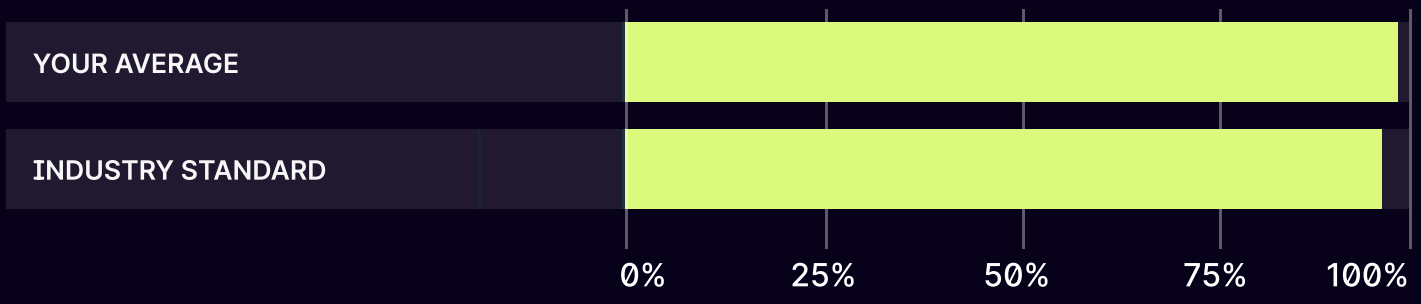
This document outlines the overall security of the Outcome smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Outcome smart contract/s codebase for quality, security, and correctness.

## Contract Status


There were 2 critical issues found during the review. (See Complete Analysis)

## Testable Code




100% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Outcome team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

# Table of Contents

# AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Outcome repository:

Repo: https://github.com/InfiniteCode-Org/outcome-market-smart-contract
Initial commit - 49efe68104b53521ddce94627884e6e92f72eb79
Last commit - b2c4f1349844e7cd8829f8911ea7375c94bd17da

**Contracts under the scope:**

- ./MarketImplementation.sol
- ./Oracle.sol
- ./OutcomeToken1155.sol
- ./MarketFactory.sol
- ./IMarket.sol

Repo: https://github.com/InfiniteCode-Org/outcome-matching-engine-smart-contract
Initial commit - 06cc1183451b749c634b29d3ed28782671efb5f3
Last commit - bffcb2188f45e04366726705c1ed79c5e9b9ebe0

**Contracts under the scope:**

- ./MatchingEngine.sol

Repo: https://github.com/InfiniteCode-Org/outcome-wallet-smart-contract
Initial commit - 8e3a2bb72ec8054458074c9b8bc07dc1299e691d
Last commit - af07ad5adc5bac1e23b5d95f5b2177e62366e103

**Contracts under the scope:**

- ./WalletFactory.sol
- ./WalletBeacon.sol
- ./WalletImplementation.sol

**During the audit, Zokyo Security ensured that the contract:**

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Outcome smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Foundry testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

| | |
|---|---|
| 01 | Due diligence in assessing the overall code quality of the codebase. |
| 02 | Cross-comparison with other, similar smart contract/s by industry leaders. |
| 03 | Testing contract/s logic against common and uncommon attack vectors. |
| 04 | Thorough manual review of the codebase line by line. |

# Executive Summary

The outcome project is made up of 3 different pilars:

**Wallet:** Allows a trusted server to execute transactions on behalf of the user using personal_sign-style signatures. It supports single and batched transaction execution with nonce-based replay protection, but only the batch signature includes the chain ID. The contract handles ETH and ERC20 transfers, and includes a withdraw function with a hardcoded 1 USDC fee, assuming 6 decimals without verifying the token. ERC1155 receiving is supported. All sensitive functions are restricted to the server address, and signature verification ensures only the designated owner can authorize actions.

**Matching Engine:** Lets a server match token swap orders between two users via their wallets. Users sign off-chain orders with swap terms, and the server executes matching logic on-chain. It verifies signatures, checks order compatibility, and transfers ERC20 or ERC1155 tokens between wallets. USDC fees are optionally paid to a collector. It prevents overfilling orders and allows cancellation. There's also a variant for collateral-based markets that mints outcomes by transferring USDC. Only the server can call sensitive functions.

**Market:** Manages a prediction market's lifecycle. Admins set it up with outcome tokens, collateral, time window, and a matching engine. Users trade outcome tokens through off-chain signed orders, which the engine matches. Orders are only matched when the market is open and within time limits. An oracle or admin resolves the market to outcome 1 or 2. Users with winning tokens can claim collateral rewards by providing a valid signature. The contract tracks status, enforces roles, prevents reuse of signatures, and burns outcome tokens on reward claim.

# STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Name of company team and the Name of company team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

**Critical**

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

**High**

The issue affects the ability of the contract to compile or operate in a significant way.

**Medium**

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

**Low**

The issue has minimal impact on the contract’s ability to operate.

**Informational**

The issue has no impact on the contract’s ability to operate.

# COMPLETE ANALYSIS

## FINDINGS SUMMARY

| # | Title | Risk | Status |
|---|---|---|---|
| 1 | Market can be resolved at any time, before endTime has been reached, leading to the attacker winning every time | Critical | Resolved |
| 2 | Attacker Can Resolve A Market Twice And Hold Both Market Tokens To Steal Rewards | Critical | Resolved |
| 3 | Users Who Win A Prediction Are Not Incentivized | High | Resolved |
| 4 | Stale Price Can Be Provided To The updatePriceAndFulfill Function | High | Resolved |
| 5 | A User Can Provide Empty updateData And Pay `0` Fees To Resolve The Market | High | Resolved |
| 6 | The Winner Of The Trigger Condition EQ Would Be Almost Everytime Be Token 2 | Medium | Resolved |
| 7 | Reentrancy In matchOrders To Violate filledAmounts Restriction | Medium | Resolved |
| 8 | Missing Validation of Pyth Price, Confidence, and Exponent in getPrice() | Medium | Unresolved |
| 9 | Centralization risk | Medium | Acknowledged |
| 10 | Different markets can be created with the same ID, which may result in an incorrect burn ID when claiming rewards | Medium | Resolved |
| 11 | Single Signature Verification is Susceptible to Cross-Chain Replay Attacks | Medium | Resolved |
| 12 | Unauthorized Wallet Creation Enables Address Hijacking in createWallet() | Low | Resolved |
| 13 | The Market Can Be Resolved Even When In Emergency Situations | Low | Resolved |

| # | Title | Risk | Status |
|---|---|---|---|
| 14 | Unsafe Ether Transfer Using transfer | Informational | Acknowledged |
| 15 | Different functions produce the same error message | Informational | Resolved |
| 16 | `amount` and the actual amount transferred may not match | Informational | Resolved |
| 17 | Dangerous actions | Informational | Acknowledged |
| 18 | Admin can trigger the situation to resolve the market twice | Informational | Acknowledged |

**CRITICAL-1** | RESOLVED

**Market can be resolved at any time, before endTime has been reached, leading to the attacker winning every time.**

**Description:**
The `updatePriceAndFulfill()` function within the `Oracle.sol` smart contract is the one that checks the current spot price and decides who is the winner. However, the function lacks a critical check to verify whether it is being called after the market's endTime. This allows any user to call the function at any point, including before the market is supposed to close.

```
function updatePriceAndFulfill(
        address market,
        bytes[] calldata updateData
    ) external payable {
        require(market != address(0), "Oracle: zero market"); // @audit
it is not checked if the function is called after endTime

        /*── 1. Pay Pyth fee and publish the update
──────────────────────*/
        uint256 fee = PYTH.getUpdateFee(updateData);
        require(msg.value >= fee, "Oracle: fee too low");
        PYTH.updatePriceFeeds{value: fee}(updateData);

        /*── 2. Get market parameters in one call
─────────────────────────*/
        IMarket.TriggerCondition memory tc =
IMarket(market).getMarketParams();


        /*── 3. Fetch fresh price
──────────────────────────────────────────*/
        IPyth.Price memory p = PYTH.getPriceUnsafe(tc.assetId);
        uint256 spot = _toUint(p.price, p.expo);

        /*── 4. Decide winner
──────────────────────────────────────────────*/
```

```
        uint256 winningToken;
        if (tc.op == IMarket.Operator.LT) {
            winningToken = (spot < tc.triggerPrice) ? 1 : 2;
        } else if (tc.op == IMarket.Operator.GT) {
            winningToken = (spot > tc.triggerPrice) ? 1 : 2;
        } else { // EQ
            winningToken = (spot == tc.triggerPrice) ? 1 : 2;
        }

        /*── 5. Resolve the market
──────────────────────────────────────*/
        IMarket(market).resolveMarketOracle(winningToken);

        /*── 6. Refund any excess ETH
──────────────────────────────────────*/
        uint256 refund = msg.value - fee;
        if (refund != 0) {
            payable(msg.sender).transfer(refund);        }
    }
```

**Impact:**
A malicious user can observe the current market conditions, place a bid based on known information, and immediately call updatePriceAndFulfill() to resolve the market in their favor. Since there is no restriction enforcing that the function can only be called after the market's official end time, users can effectively guarantee a win by timing their actions—leading to unfair and potentially manipulative outcomes.

POC: https://gist.github.com/V1C70RYG0D/3447bb8847125d7ffa6c052aa19bdae9

**Recommendation:**
Add a check to ensure that updatePriceAndFulfill() can only be called after the market's endTime.

**Attacker Can Resolve A Market Twice And Hold Both Market Tokens To Steal Rewards**

**Description:**
Consider the following →

1. There's a prediction market where trigger condition as EQ , the attacker puts two orders off-chain one for each side of the prediction.
2. The market is resolved by the admin or any user at appropriate time , let's assume the price was exactly equal to attacker's prediction (triggerPrice in the TriggerCondition) , then the attacker claims the rewards using claimRewardWithSig() and gets the deserving reward i.e USDC.
3. As soon as the market price moves even with 1wei , the attacker calls updatePriceAndFull() again and this time the winning token would be other token i.e. winning outcome would be 2 →

```
uint256 winningToken;
        if (tc.op == IMarket.Operator.LT) {
            winningToken = (spot < tc.triggerPrice) ? 1 : 2;
        } else if (tc.op == IMarket.Operator.GT) {
            winningToken = (spot > tc.triggerPrice) ? 1 : 2;
        } else { // EQ
            winningToken = (spot == tc.triggerPrice) ? 1 : 2;
        }
```

4. This time when resolve is invoked in the MarketImplementation.sol contract it does not check if the market is previously resolved , it assigns the winning outcome as 2 and again assigns the status as RESOLVED.
5. The attacker claims the reward again using claimRewardWithSig() but this time claims with the other outcome token Id , therefore stealing rewards of honest users who bet on the EQ side of the market.

**Impact:**
A prediction market can be resolved twice with different outcomes leading to the attacker stealing honest user's rewards.

**Recommendation:**
Once resolved don't let the market be resolved again.

**Users Who Win A Prediction Are Not Incentivized**

**Description:**
Consider the following case →

1. UserA puts up an off-chain order of `100` USDC and `100` outcome tokens (tokenId1) , when the order is matched then UserA gets `100` outcome tokens.
2. When the prediction is resolved by the pyth oracle the winning outcome is set in the resolveMarketOracle() function.
3. UserA can now claim his reward using the claimRewardWithSig function , the outcome tokens are burnt and then equivalent amount of USDC is transferred to the user i.e. only `100` USDC is transferred to the user which is `100` wei of USDC and the user actually incurs a loss here.

The correct approach should be that the rewards should be distributed proportionally to the balance of the outcome tokens of the user and the total supply of winning outcome tokens and multiply that with the total balance of USDC in the Market contract.

**Impact:**
The reward calculation would reward winners incorrectly.

**Recommendation:**
Distribute reward proportionally , for example → rewardUSDC = totalUSDC * (winningTokenBalanceOfUser) / totalSupplyOfWinningToken

**Stale Price Can Be Provided To The updatePriceAndFulfill Function**

**Description:**
When providing the pyth report to the function updatePriceAndFulfill it is not checked if the price report is fresh , that is , when the price is fetched using IPyth.Price memory p = PYTH.getPriceUnsafe(tc.assetId); , it is not verified if the p.publishTime is fresh enough or is not stale , this means that any user can provide an outdated but valid price report and that price would be used to determine the winning outcome token.

**Impact:**
The winning outcome of the market would be determined by a stale price from pyth oracle.

**Recommendation:**
Verify that the price is not stale.

**A User Can Provide Empty updateData And Pay 0 Fees To Resolve The Market**

**Description:**
When a user calls updatePriceAndFulfill() he must pass updateData which is used to update the price feed , but a user can pass empty updateData and when updatePriceFeeds is called on pyth →

```
function updatePriceFeeds(
    bytes[] calldata updateData
) public payable override {
    uint totalNumUpdates = 0;
    for (uint i = 0; i < updateData.length; ) {
        if (
            updateData[i].length > 4 &&
            UnsafeCalldataBytesLib.toUint32(updateData[i], 0) ==
            ACCUMULATOR_MAGIC
        ) {
            totalNumUpdates += updatePriceInfosFromAccumulatorUpdate(
                updateData[i]
            );
        } else {
            updatePriceBatchFromVm(updateData[i]);
            totalNumUpdates += 1;
        }

        unchecked {
            i++;
        }
    }
    uint requiredFee = getTotalFee(totalNumUpdates);
    if (msg.value < requiredFee) revert PythErrors.InsufficientFee();
}
```

Meaning it would skip the update and required fees would be zero plus the price would be fetched from unsafe method instead of the getPrice() method in Pyth oracle.

**Impact:**
An Attacker can bypass the update fee plus the price is fetched from the unsafe method instead of the update data.

**Recommendation:**
Ensure that updateData.length > 0.

**MEDIUM-1** | RESOLVED

**The Winner Of The Trigger Condition EQ Would Be Almost Everytime Be Token 2**

**Description:**
If the trigger condition is EQ then the price fetched from the pyth oracle should be EXACTLY equal to the trigger price i.e. it should be accurate up to 8 decimals , this magnitude of precision would be almost impossible to maintain , even if the price deviates by 1 basis point (magnitude of 1/1e8) the winner would be token 2 and it is almost impossible that the fetched price lands exactly at the triggered price.

**Impact:**
Outcome Token1 holders would almost never win due to extreme specific trigger price condition.

**Recommendation:**
For such markets we should have an accepted deviation percentage amount around which outcome token 1 wins and else token 2 wins.

**Reentrancy In matchOrders To Violate filledAmounts Restriction**

**Description:**
When matchOrders() is called in the MatchingEngine.sol it is ensured that we don't overfill the orders →

```
require(
            filledAmounts[hA] + fillAmountA <= orderA.amountIn &&
            filledAmounts[hB] + fillAmountB <= orderB.amountIn,
            "overfill"
        );
```

And then the tokens are transferred , token can be either the collateral token or the ERC1155 token , in case of ERC1155 since safeTransferFrom is used it would call the onERC1155Received() hook of the receiver which can again invoke the matchOrders function (provided the receiver is the permitted server address) , the orders would be matched again and the overfill condition will still satisfy but the filledAmounts mapping is only updated at the end of the function and it would end up filling twice more than it should.

**Impact:**
filledAmount would be far greater than the actual amount requested in the order.

**Recommendation:**
Update the filledAmounts mapping before transferring the tokens.

**Missing Validation of Pyth Price, Confidence, and Exponent in getPrice()**

**Description:**
The contract uses PYTH.getPrice() to retrieve price data, but fails to validate the returned price, confidence interval (conf), and expo values before using them. This exposes the contract to the risk of accepting invalid or untrusted prices, which could result in incorrect logic execution, especially in critical functions like market resolution.
According to the Pyth documentation, it is crucial to validate the confidence interval (conf) to ensure that price data is reliable. Additionally, checking for non-positive prices and unrealistic exponents (e.g., expo < -18) prevents obviously corrupted or meaningless values from being used.

**Impact:**
If no validation is in place, the system may:

- Accept outdated or manipulated prices with large uncertainty.
- Resolve markets or execute logic based on meaningless data.
- Be vulnerable to oracle manipulation attacks where invalid inputs appear legitimate.

**Recommendation:**
- Always validate price > 0
- Ensure expo is within a reasonable range (e.g., not less than -18)
- Check the confidence interval using a MIN_CONFIDENCE_RATIO to reject untrusted prices
- Follow Pyth's best practices for secure oracle usage

```
if (price <= 0 || expo < -18) {
 revert("PA:getAssetPrice:INVALID_PRICE");
}

if (conf > 0 && (price / int64(conf) < MIN_CONFIDENCE_RATIO)) {
rever
```

**Centralization risk**

**Description:**
This issue describe the centralized nature of the backend behavior. The backend is a really important operator on the whole protocol as it is the one executing certain actions.
This issue has been created to encompass all issues related to the centralized nature of the backend.

1. The `initialize()` function within the MarketImplementation.sol smart contract implements the following require statement to allow the execution:

require(status == MarketStatus(0) && marketId == 0, "init");

However, it does not check if the marketId is going to be set to 0.
The severity of this issue has been set as Medium because the condition of the owner setting a market implementation with marketId to 0 needs to be met.

If the MarketImplementation is initialized with marketId to 0 it allows any user to call initialize() again and update any important parameters like the admin control of the contract.

POC: https://gist.github.com/SumitZokyo/fb49e06bdd170649800aad09f1170ddf

2. When off-chain orders are matched with matchOrdersAndMint() it is not verified that these orders are for different tokenIds i.e. one is for outcome as 1 and the other for outcome as 2 , if these orders are of the same type then the matching would be incorrect as per the prediction logic , moreover it should be checked that the tokenIds is a valid tokenId which is derived from the marketId as marketId * 2 and marketId * 2 + 1.

3. The `withdraw()` function within the `WalletImplementation.sol` smart contract is used to withdraw tokens enforcing 1 USDC fee. The USDC fee is hardcoded within `USDC_DECIMALS` to `1e6`, however, the token amount is not set within the contract

address but used as a function parameter. This may lead to a scenario where the token used is not 6 decimals.
If a token with a decimal count other than 6 is used in the `withdraw()` function, the fee amount received may not match the expected 1 USDC.

POC: https://gist.github.com/SumitZokyo/a1c12ae585579f031b92f8def5369e5d

4. The `matchOrders()` function within the `MatchingEngine.sol` smart contract receives 2 parameters defining the fees to each of the users. These fees are directly selected by the server and do not have an upper limit. It is also important to note that fees should be the same for each user.

5. A malicious user can put up an order of 1 USDC and 100 outcome tokens and another honest user can put up an order of 100 USDC and 100 outcome tokens , if these orders are matched with the matchOrdersAndMint() function then the malicious user is exposed to the winning side of the market with 1/100th cost relative to the honest user with equal reward opportunity.

**Different markets can be created with the same ID, which may result in an incorrect burn ID when claiming rewards.**

**Description:**
The `createMarket()` function within the `MarketFactory.sol` smart contract is used to deploy a new MarketImplementation and initialize it.
This function receives several parameters that are used to initialize the created market:

```
function createMarket(
        address _admin,
        address _oracle,
        address _collateralToken,
        address _outcomeToken1155,
        address _walletFactory,
        uint256 _marketId,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _chainId,
        IMarket.TriggerCondition calldata tc
    ) external onlyOwner returns (address) {
        require(_admin != address(0), "Invalid admin");
        require(_oracle != address(0), "Invalid oracle");
        require(_collateralToken != address(0), "Invalid collateral");
        require(_outcomeToken1155 != address(0), "Invalid outcome
token");
        require(_walletFactory != address(0), "Invalid wallet factory");
        require(_endTime > _startTime, "Invalid time range");

        // Deploy a fresh MarketImplementation
        MarketImplementation market = new MarketImplementation();


        // Initialize
        market.initialize(
            _admin,
```

```
            _oracle,
            _collateralToken,
            _outcomeToken1155,
            _walletFactory,
            _marketId,
            _startTime,
            _endTime,
            _chainId,
            matchingEngine,
            tc
        );

        // Authorize the new market to mint from the outcome token (if
needed).
        OutcomeToken1155(_outcomeToken1155).setMinter(address(market),
true);

        allMarkets.push(address(market));
        marketAddress[_marketId] = address(market);
        emit MarketCreated(address(market), _marketId, _admin, _oracle);
        return address(market);
    }
```

As it can be observed, the marketId is the id defining each market and is used override the marketAddress mapping while there are no requirements to not reuse past market ids:

```
marketAddress[_marketId] = address(market);
```

The severity of this issue has been market as medium because the function is access restricted, otherwise, if any user could create markets freely, selecting the marketId, the severity would have been higher.

**Impact:**
If two different markets are created with the same `marketId` then the `marketAddress` will be overridden, pointing to the last included one.

Moreover, the tokenId burnt when a user claims rewards from a market is selected using this marketId:

```
uint256 winId = _tokenIdForOutcome(winningOutcome);
        uint256 bal   = outcomeToken.balanceOf(wallet, winId);
        require(bal > 0, "bal 0");

        outcomeToken.burn(wallet, winId, bal);


function _tokenIdForOutcome(uint256 o) internal view returns (uint256) {
        return (o == 1) ? marketId * 2 : marketId * 2 + 1;
    }
```

POC: https://gist.github.com/SumitZokyo/fcb3682c47cc8647040f374b75f24dcb

**Recommendation:**
Include a mapping within `MarketFactory.sol` to record the already used marketIds and implement a require statement to not allow new markets being created with the same marketId than previous ones.

**Single Signature Verification is Susceptible to Cross-Chain Replay Attacks**

**Description:**
The `_verifySignature()` function within the `WalletImplementation.sol` smart contract is used to verify is the signature used is correct to allow executing the action or not. The hash used contains arguments like `target`, `value`, `data` and `nonce`. However, it does not include `chainId`. This means that the same signature can be used twice in different chains if the protocol is deployed in different ones at any point:

```
function _verifySignature(
        address target,
        uint256 value,
        bytes calldata data,
        bytes calldata signature
    ) internal {
        bytes32 messageHash = keccak256(abi.encodePacked(target, value,
data, nonce));
        bytes32 ethSignedMessage = keccak256(
            abi.encodePacked("\x19Ethereum Signed Message:\n32",
messageHash)
        );
        address recovered = ECDSA.recover(ethSignedMessage, signature);
      require(recovered == owner, "Invalid signature");
        nonce++;

    }
```

**Impact:**
If the described scenario takes place, the same signature generated to be executable for 1 chain can be replayed in another chain, leading to 2 different transactions being executed unexpectedly.

**Recommendation:**
In order to avoid cross chain replay attacks, add the chainId as a hash parameter as it is being done within the `verifyBatchSignature()` function.

LOW-1 | RESOLVED

**Unauthorized Wallet Creation Enables Address Hijacking in createWallet()**

**Description:**
The `createWallet()` function in the `WalletFactory.sol` smart contract allows any external user to create a wallet for a specified owner address. This is due to the function being marked as external and accepting owner as a parameter, enabling arbitrary input. Additionally, the deterministic nature of the wallet address (controlled by the salt) allows an attacker to predict and preemptively deploy a wallet on behalf of another user.

```
require(ownerToWallet[owner] == address(0), "Owner already has a
wallet");
```

**Impact:**
Given that only one wallet can be created per owner, this can effectively block legitimate users from creating their own wallet via the factory.

This means that if a user attempts to create a wallet for a specific address, the operation can be preempted by another user who creates a wallet for that same address first, even by front-running the original transaction.

**Recommendation:**
Modify the `createWallet()` function to use `msg.sender` as the wallet owner instead of receiving `owner()` as function parameter.

LOW-2 | RESOLVED

**The Market Can Be Resolved Even When In Emergency Situations**

**Description:**
In emergency situations the market would be paused using the pauseMarket() function , but even in these emergency situations price can be fetched from the pyth oracle and the market can be resolved and rewards can be claimed using the resolveMarketOracle() and claimRewardWithSig() functions.

**Recommendation:**
If the market is paused then also pause the resolveMarketOracle() with the whenNotPaused modifier.

INFORMATIONAL-1 | ACKNOWLEDGED

**Unsafe Ether Transfer Using transfer**

**Description:**
The `updatePriceAndFulfill()` function within the `Oracle.sol` smart contract uses payable(msg.sender).transfer(refund); to send Ether. This approach is considered unsafe because the transfer function forwards only 2300 gas, which might not be sufficient if the recipient is a smart contract with a fallback function that requires more gas. This could cause transactions to fail unexpectedly and potentially lock funds.

**Impact:**
It can cause transactions of ether fail unexpectedly.

**Recommendation:**
Use call instead of transfer for sending Ether:

```
(bool success, ) = msg.sender.call{value: refund}("");
require(success, "Refund failed");
```

## INFORMATIONAL-2 | RESOLVED

**Different functions produce the same error message**

**Description:**
The `changeServer()` and `changeFeeCollector()` functions within the `MatchingEngine.sol` smart contract emit the same error when the new address is set to `0`.

**Impact:**
There are no security implications here, but it is recommended to have unique error messages in order to identify where the error has been produced in case that a transaction is reverted.

**Recommendation:**
Add unique error messages for each function.

## INFORMATIONAL-3 | RESOLVED

**`amount` and the actual amount transferred may not match**

**Description:**
The `withdraw()` function within the `WalletImplementation.sol` contract receives `amount` as argument, which is expected to represent the amount of tokens that are going to be withdrawn.
However, this amount is taken from the `data` not from the `amount` variable.

## INFORMATIONAL-4 | ACKNOWLEDGED

**Dangerous actions**

**Description:**
The ´WalletImplementation.sol` smart contract allow users to execute several functions which execute arbitrary data to a selected target contract. While these are the expected features of the contract, it is important to mark that these actions can lead to dangerous scenarios.

## INFORMATIONAL-5 | ACKNOWLEDGED

**Admin can trigger the situation to resolve the market twice**

**Description:**
Related to issue number 2, which has been resolved, it is still possible to resolve the market twice if after the first time resolving, the admin executes `unpauseMarket()`. This will set the MarketStatus to `OPEN` again, leading to the possibility of resolving for the second time.

This issue has been included as informational as it is expected that the admin is a trusted entity that will not execute the mentioned flow but it is important to mention it as it is technically possible.

| | ./MarketImplementation.sol<br>./Oracle.sol<br>./OutcomeToken1155.sol<br>./MarketFactory.sol<br>./IMarket.sol |
|---|---|
| Re-entrancy | Pass |
| Access Management Hierarchy | Pass |
| Arithmetic Over/Under Flows | Pass |
| Unexpected Ether | Pass |
| Delegatecall | Pass |
| Default Public Visibility | Pass |
| Hidden Malicious Code | Pass |
| Entropy Illusion (Lack of Randomness) | Pass |
| External Contract Referencing | Pass |
| Short Address/ Parameter Attack | Pass |
| Unchecked CALL<br>Return Values | Pass |
| Race Conditions / Front Running | Pass |
| General Denial Of Service (DOS) | Pass |
| Uninitialized Storage Pointers | Pass |
| Floating Points and Precision | Pass |
| Tx.Origin Authentication | Pass |
| Signatures Replay | Pass |
| Pool Asset Security (backdoors in the underlying ERC-20) | Pass |

| | ./MatchingEngine.sol<br>./WalletFactory.sol<br>./WalletBeacon.sol<br>./WalletImplementation.sol |
|---|---|
| Re-entrancy | Pass |
| Access Management Hierarchy | Pass |
| Arithmetic Over/Under Flows | Pass |
| Unexpected Ether | Pass |
| Delegatecall | Pass |
| Default Public Visibility | Pass |
| Hidden Malicious Code | Pass |
| Entropy Illusion (Lack of Randomness) | Pass |
| External Contract Referencing | Pass |
| Short Address/ Parameter Attack | Pass |
| Unchecked CALL<br>Return Values | Pass |
| Race Conditions / Front Running | Pass |
| General Denial Of Service (DOS) | Pass |
| Uninitialized Storage Pointers | Pass |
| Floating Points and Precision | Pass |
| Tx.Origin Authentication | Pass |
| Signatures Replay | Pass |
| Pool Asset Security (backdoors in the underlying ERC-20) | Pass |

# CODE COVERAGE AND TEST RESULTS FOR ALL FILES

## Tests written by Zokyo Security

As a part of our work assisting Outcome in verifying the correctness of their contract/s code, our team was responsible for writing integration tests using the Foundry testing framework.

The tests were based on the functionality of the code, as well as a review of the Outcome contract/s requirements for details about issuance amounts and how the system handles these.

**Ran 40 tests for test/foundry/MarketFactory.t.sol:MarketFactoryTest**
[PASS] testConstructorSetsCorrectValues() (gas: 19410)
[PASS] testConstructorWithZeroAddress() (gas: 4359200)
[PASS] testConstructorWithZeroOwner() (gas: 45942)
[PASS] testCreateMarket() (gas: 4103067)
[PASS] testCreateMarketEmitsEvent() (gas: 4088408)
[PASS] testCreateMarketEventEmissionOrder() (gas: 4106720)
[PASS] testCreateMarketGasEfficiency() (gas: 8101349)
[PASS] testCreateMarketRevertInvalidTimeRange() (gas: 62413)
[PASS] testCreateMarketRevertMarketIdExists() (gas: 4085678)
[PASS] testCreateMarketRevertNotOwner() (gas: 53093)
[PASS] testCreateMarketRevertZeroAdmin() (gas: 52524)
[PASS] testCreateMarketRevertZeroCollateral() (gas: 52590)
[PASS] testCreateMarketRevertZeroOracle() (gas: 52534)
[PASS] testCreateMarketRevertZeroOutcomeToken() (gas: 41656)
[PASS] testCreateMarketRevertZeroWalletFactory() (gas: 52613)
[PASS] testCreateMarketWithAllOperatorTypes() (gas: 12104938)
[PASS] testCreateMarketWithDifferentAdminAndOracle() (gas: 4088181)
[PASS] testCreateMarketWithDifferentChainIds() (gas: 12116946)
[PASS] testCreateMarketWithDifferentChainIdsExtensive() (gas: 20165369)
[PASS] testCreateMarketWithDifferentTriggerConditions() (gas: 8080809)
[PASS] testCreateMarketWithExactMinimumTimeGap() (gas: 4084963)
[PASS] testCreateMarketWithExtremeTimeValues() (gas: 8101717)
[PASS] testCreateMarketWithExtremeTriggerPrices() (gas: 8098815)
[PASS] testCreateMarketWithManyTriggerConditions() (gas: 40147471)
[PASS] testCreateMarketWithMaximumParameters() (gas: 4060363)
[PASS] testCreateMarketWithSameId() (gas: 8087948)
[PASS] testCreateMultipleMarkets() (gas: 12129352)
[PASS] testFactoryStateAfterManyMarkets() (gas: 201101732)
[PASS] testFuzz_CreateMarketWithDifferentIds(uint256) (runs: 256, μ: 4087300, ~: 4087300)

[PASS] testFuzz_CreateMarketWithDifferentOperators(uint256,uint8) (runs: 256, μ: 6140244, ~: 6144056)
[PASS] testFuzz_CreateMarketWithDifferentTimeRanges(uint64,uint64) (runs: 256, μ: 4085703, ~: 4085703)
[PASS] testFuzz_CreateMarketWithDifferentTriggerPrices(uint256) (runs: 256, μ: 4085088, ~: 4084465)
[PASS] testFuzz_CreateMarketWithInvalidParameters(address[],uint256,uint256) (runs: 256, μ: 10070, ~: 9970)
[PASS] testFuzz_CreateMarketWithVaryingParameters(uint256,uint256,uint256,uint256) (runs: 256, μ: 6159977, ~: 6159871)
[PASS] testFuzz_CreateMarketsWithDifferentOperators(uint256,uint8) (runs: 256, μ: 6140243, ~: 6144055)
[PASS] testFuzz_CreateMultipleMarketsSequentially(uint8) (runs: 256, μ: 23640737, ~: 18333302)
[PASS] testGetAllMarketsEmptyArray() (gas: 11939)
[PASS] testGetAllMarketsNonEmptyArray() (gas: 8104743)
[PASS] testInitialState() (gas: 24071)
[PASS] testOwnerTransferOwnership() (gas: 22277)
Suite result: ok. 40 passed; 0 failed; 0 skipped; finished in 1.76s (5.04s CPU time)

**Ran 52 tests for test/foundry/MarketImplementation.t.sol:MarketImplementationTest**
[PASS] testClaimRewardWithDifferentNonces() (gas: 291178)
[PASS] testClaimRewardWithInvalidSignature() (gas: 134893)
[PASS] testClaimRewardWithSig() (gas: 247625)
[PASS] testClaimRewardWithSigRevertInvalidSignature() (gas: 187661)
[PASS] testClaimRewardWithSigRevertMarketNotResolved() (gas: 129717)
[PASS] testClaimRewardWithSigRevertNoWinOutcome() (gas: 122545)
[PASS] testClaimRewardWithSigRevertNonceUsed() (gas: 228831)
[PASS] testClaimRewardWithSigRevertNullWallet() (gas: 110520)
[PASS] testClaimRewardWithSigRevertZeroBalance() (gas: 143297)
[PASS] testClaimRewardWithSigRevertZeroWallet() (gas: 69666)
[PASS] testClaimStructAccessibility() (gas: 4207)
[PASS] testConstantValues() (gas: 12364)
[PASS] testDomainSeparatorUniqueness() (gas: 7901568)
[PASS] testEventEmissionForAllActions() (gas: 258299)
[PASS] testFallbackResolutionExactTiming() (gas: 4032978)
[PASS] testFuzz_ClaimRewardWithDifferentAmounts(uint256) (runs: 256, μ: 225753, ~: 225753)
[PASS] testFuzz_ClaimRewardWithDifferentOutcomes(uint256) (runs: 256, μ: 225046, ~: 225075)
[PASS] testFuzz_EIP712SignatureValidation(uint256) (runs: 256, μ: 215814, ~: 215415)
[PASS] testFuzz_MarketLifecycle(uint256,uint256,uint256) (runs: 256, μ: 10126974, ~: 10126882)
[PASS] testFuzz_MarketResolutionWithDifferentOutcomes(uint256) (runs: 256, μ: 220594, ~: 220623)
[PASS] testFuzz_OrderMatching(uint256,uint256,uint256,uint256) (runs: 256, μ: 143097, ~: 142254)

[PASS] testFuzz_OrderSignatureVerification(uint256,uint256,uint256,uint8,uint256,uint256) (runs: 256, μ: 177058, ~: 177004)
[PASS] testGetMarketParams() (gas: 24341)
[PASS] testInitializeRevertAlreadyInitialized() (gas: 45635)
[PASS] testInitializeRevertInvalidTimeRange() (gas: 3442370)
[PASS] testInitializeSetsCorrectValues() (gas: 69938)
[PASS] testInitializeWithExtremeValues() (gas: 3941682)
[PASS] testMarketLifecycleComplexTransitions() (gas: 123140)
[PASS] testMatchOrdersAndMint() (gas: 133624)
[PASS] testMatchOrdersAndMintRevertNoEngine() (gas: 44749)
[PASS] testMatchOrdersAndMintRevertNullEngine() (gas: 45104)
[PASS] testMatchOrdersAndMintRevertOutsideTimeWindow() (gas: 61820)
[PASS] testMatchOrdersAndMintRevertWhenPaused() (gas: 83353)
[PASS] testMatchOrdersWithDifferentTokenTypes() (gas: 133076)
[PASS] testOrderStructAccessibility() (gas: 10474)
[PASS] testPauseMarket() (gas: 65865)
[PASS] testPauseMarketRevertNotAdmin() (gas: 14156)
[PASS] testResolveMarketBothOutcomes() (gas: 7996452)
[PASS] testResolveMarketFallback() (gas: 71542)
[PASS] testResolveMarketFallbackRevertNotAdmin() (gas: 18685)
[PASS] testResolveMarketFallbackRevertTooEarly() (gas: 21017)
[PASS] testResolveMarketOracle() (gas: 65319)
[PASS] testResolveMarketOracleRevertInvalidOutcome() (gas: 14753)
[PASS] testResolveMarketOracleRevertNotOracle() (gas: 14727)
[PASS] testRoleAssignmentAndRetrieval() (gas: 48628)
[PASS] testSetEngine() (gas: 21834)
[PASS] testSetEngineRevertNotAdmin() (gas: 14668)
[PASS] testSetEngineRevertNotAdminFromNonAdmin() (gas: 14714)
[PASS] testTokenIdCalculation() (gas: 18462505)
[PASS] testTokenIdForOutcome() (gas: 277067)
[PASS] testUnpauseMarket() (gas: 52572)
[PASS] testUnpauseMarketRevertNotAdmin() (gas: 66435)
Suite result: ok. 52 passed; 0 failed; 0 skipped; finished in 52.17s (3.92s CPU time)

**Ran 43 tests for test/foundry/OutcomeToken1155.t.sol:OutcomeToken1155Test**
[PASS] testBatchOperations() (gas: 455877)
[PASS] testBurnAsAuthorizedMinter() (gas: 81265)
[PASS] testBurnEmitsTransferSingleEvent() (gas: 81952)
[PASS] testBurnExceedingBalance() (gas: 81930)
[PASS] testBurnFullBalance() (gas: 61124)
[PASS] testBurnMultipleTokens() (gas: 161020)
[PASS] testBurnNonexistentTokens() (gas: 52547)

[PASS] testBurnRevertNotAuthorized() (gas: 79047)
[PASS] testComplexBurningScenarios() (gas: 95210)
[PASS] testComplexMinterManagement() (gas: 648958)
[PASS] testConcurrentMinterOperations() (gas: 155072)
[PASS] testConstructorWithZeroAddresses() (gas: 63884)
[PASS] testExtremeMintingScenarios() (gas: 131440)
[PASS] testFuzz_BurnPartialBalance(uint256,uint256) (runs: 256, μ: 85403, ~: 86701)
[PASS] testFuzz_MintAndBurn(uint256) (runs: 256, μ: 84870, ~: 84870)
[PASS] testFuzz_MintBurnSequence(address,uint256,uint256,uint256) (runs: 256, μ: 126251, ~: 123579)
[PASS] testFuzz_MintMultipleTokenIds(uint256,uint256) (runs: 256, μ: 79483, ~: 79577)
[PASS] testFuzz_MinterAuthorizationInvariant(address,address[]) (runs: 256, μ: 17271, ~: 17043)
[PASS] testFuzz_MinterAuthorizationSequence(address[],bool[]) (runs: 256, μ: 142058, ~: 139020)
[PASS] testFuzz_MultipleMintersCompetition(address[],uint256[][],uint256) (runs: 256, μ: 266101, ~: 262034)
[PASS] testFuzz_SetMinterMultipleAddresses(address,bool) (runs: 256, μ: 29811, ~: 39373)
[PASS] testInitialState() (gas: 37309)
[PASS] testInterfaceSupport() (gas: 16278)
[PASS] testMintAsAuthorizedMinter() (gas: 106693)
[PASS] testMintAsOwner() (gas: 71456)
[PASS] testMintEmitsTransferSingleEvent() (gas: 74330)
[PASS] testMintMultipleTokensToSameUser() (gas: 140104)
[PASS] testMintRevertNotAuthorized() (gas: 18237)
[PASS] testMintToMultipleUsers() (gas: 109198)
[PASS] testMintWithCustomData() (gas: 74160)
[PASS] testMinterAuthorizationStatePersistence() (gas: 101134)
[PASS] testMinterPermissionHierarchy() (gas: 201536)
[PASS] testMinterRevocationDuringOperations() (gas: 66305)
[PASS] testMintingWithLargeData() (gas: 543731)
[PASS] testOwnershipTransferAndMinterPersistence() (gas: 116541)
[PASS] testSetMinter() (gas: 32259)
[PASS] testSetMinterRevertNotOwner() (gas: 17814)
[PASS] testSetMinterSelf() (gas: 38544)
[PASS] testSetMinterToCurrentValue() (gas: 60700)
[PASS] testSetMultipleMinters() (gas: 98083)
[PASS] testTokenBalanceConsistency() (gas: 152533)
[PASS] testURIFunctionality() (gas: 31056)
[PASS] testZeroAddressInteractions() (gas: 80044)
Suite result: ok. 43 passed; 0 failed; 0 skipped; finished in 52.17s (54.40s CPU time)

**Ran 57 tests for test/foundry/Oracle.t.sol:OracleTest**
[PASS] testConstructorRevertsOnZeroOwnerAddress() (gas: 41079)

[PASS] testConstructorRevertsOnZeroPythAddress() (gas: 64730)
[PASS] testFuzz_ToUint(int64,int32) (runs: 256, μ: 1103910, ~: 1105234)
[PASS] testFuzz_UpdatePriceAndFulfill_DifferentFees(uint256) (runs: 256, μ: 116455, ~: 116528)
[PASS] testFuzz_UpdatePriceAndFulfill_DifferentOperators(int64,uint8) (runs: 256, μ: 134531, ~: 133995)
[PASS] testFuzz_UpdatePriceAndFulfill_ExcessFeeRefund(uint256,uint256) (runs: 256, μ: 127807, ~: 127944)
[PASS] testFuzz_UpdatePriceAndFulfill_GT(int64) (runs: 256, μ: 128722, ~: 128722)
[PASS] testInitialState() (gas: 17512)
[PASS] testMarketResolveFailurePropagation() (gas: 82243)
[PASS] testMultipleMarketResolutions() (gas: 2156242)
[PASS] testOracleConstructorZeroOwner() (gas: 41043)
[PASS] testOracleConstructorZeroPythAddress() (gas: 62544)
[PASS] testOwnershipFunctionality() (gas: 22198)
[PASS] testPriceConversionAccuracy() (gas: 1097239)
[PASS] testPythUpdateFailurePropagation() (gas: 67444)
[PASS] testToUintBoundaryValues() (gas: 1096894)
[PASS] testToUintConsistencyAcrossValues() (gas: 1099078)
[PASS] testToUintWithExtremeNegativeExponents() (gas: 1096987)
[PASS] testToUintWithVeryLargePositiveExponents() (gas: 1100611)
[PASS] testToUint_BasicConversion() (gas: 1090297)
[PASS] testToUint_DifferentExponents() (gas: 1096943)
[PASS] testToUint_EdgeCases() (gas: 1093929)
[PASS] testToUint_ExtremelyLargePositiveExponent() (gas: 1090387)
[PASS] testToUint_MinimumPositivePrice() (gas: 1090465)
[PASS] testToUint_PositiveExponent() (gas: 1090363)
[PASS] testToUint_RevertsWithNegativePrice() (gas: 1089716)
[PASS] testToUint_RevertsWithZeroPrice() (gas: 1089694)
[PASS] testUpdatePriceAndFulfillGasOptimization() (gas: 104653)
[PASS] testUpdatePriceAndFulfillPriceComparisonEdgeCases() (gas: 1181132)
[PASS] testUpdatePriceAndFulfillRefundCalculation() (gas: 672735)
[PASS] testUpdatePriceAndFulfillSequentia[PASS] testUpdatePriceAndFulfillWithComplexUpdateData() (gas: 113029)
[PASS] testUpdatePriceAndFulfillWithDifferentAssetIds() (gas: 1737676)
[PASS] testUpdatePriceAndFulfillWithEdgeCasePrices() (gas: 1089652)
[PASS] testUpdatePriceAndFulfillWithLargeExponents() (gas: 657489)
[PASS] testUpdatePriceAndFulfillWithVeryHighFee() (gas: 111070)
[PASS] testUpdatePriceAndFulfillWithZeroUpdateFee() (gas: 91989)
[PASS] testUpdatePriceAndFulfill_BytesArrayPythUpdate() (gas: 106623)
[PASS] testUpdatePriceAndFulfill_EQ_PriceIsEqual() (gas: 120668)
[PASS] testUpdatePriceAndFulfill_EQ_PriceIsNotEqual() (gas: 123546lCalls() (gas: 655808)

[PASS] testUpdatePriceAndFulfill_ExactFeeNoRefund() (gas: 105333)
[PASS] testUpdatePriceAndFulfill_ExactlyEqualPrices() (gas: 120647)
[PASS] testUpdatePriceAndFulfill_GT_PriceIsHigher() (gas: 120777)
[PASS] testUpdatePriceAndFulfill_GT_PriceIsLower() (gas: 120810)
[PASS] testUpdatePriceAndFulfill_LT_PriceIsHigher() (gas: 128182)
[PASS] testUpdatePriceAndFulfill_LT_PriceIsLower() (gas: 128218)
[PASS] testUpdatePriceAndFulfill_NoRefundWhenExactFee() (gas: 106929)
[PASS] testUpdatePriceAndFulfill_RefundsExcessETH() (gas: 110891)
[PASS] testUpdatePriceAndFulfill_RevertsOnFeeTooLow() (gas: 34097)
[PASS] testUpdatePriceAndFulfill_RevertsOnMarketResolveFailure() (gas: 82244)
[PASS] testUpdatePriceAndFulfill_RevertsOnNegativePrice() (gas: 71728)
[PASS] testUpdatePriceAndFulfill_RevertsOnPythUpdateFailure() (gas: 61267)
[PASS] testUpdatePriceAndFulfill_RevertsOnZeroMarket() (gas: 19712)
[PASS] testUpdatePriceAndFulfill_RevertsOnZeroPrice() (gas: 71730)
[PASS] testUpdatePriceAndFulfill_WithEmptyUpdateData() (gas: 104071)
[PASS] testUpdatePriceAndFulfill_WithMultipleUpdateDataItems() (gas: 111606)
[PASS] testUpdatePriceAndFulfill_WithNonEmptyUpdateData() (gas: 106619)
Suite result: ok. 57 passed; 0 failed; 0 skipped; finished in 52.17s (1.51s CPU time)

**Ran 4 test suites in 52.20s (158.29s CPU time): 192 tests passed, 0 failed, 0 skipped (192 total tests)**

**Ran 56 tests for test/MatchingEngine.t.sol:MatchingEngineTest**
[PASS] testAdminFunctionsCoverage() (gas: 62636)
[PASS] testAdminFunctionsZeroAddressBranches() (gas: 39318)
[PASS] testConstructor() (gas: 37825)
[PASS] testConstructorSpecificValidationBranches() (gas: 414272)
[PASS] testConstructorZeroAddresses() (gas: 282189)
[PASS] testContractInteractionEdgeCases() (gas: 94944)
[PASS] testEventEmissionEdgeCases() (gas: 311612)
[PASS] testExtremeFeeSizes() (gas: 556840)
[PASS] testExtremeValues() (gas: 2425271)
[PASS] testFuzz_ChainReorgScenario(uint256,uint256,uint256,uint256,bool) (runs: 260, μ: 2379243, ~: 2439793)
[PASS] testFuzz_ComplexPartialFills(uint256,uint256,uint8,uint8) (runs: 260, μ: 2674348, ~: 2660448)
[PASS] testFuzz_ConcurrentOrderOperations(uint256,uint256,uint256,uint256,bool) (runs: 260, μ: 2563148, ~: 2563689)
[PASS] testFuzz_CrossTokenTypeMatching(uint256,uint256,uint256) (runs: 260, μ: 1739271, ~: 1739993)
[PASS] testFuzz_ERC1155TokenIdBoundaries(uint256,uint256,uint256,uint256) (runs: 260, μ: 360173, ~: 361165)

[PASS] testFuzz_FeesAndFeeRecipient(uint256,uint256,uint256,uint256,address) (runs: 260, μ: 2775681, ~: 2775821)
[PASS] testFuzz_MatchOrdersAndMintRatios(uint256,uint256,uint256,uint256) (runs: 260, μ: 308675, ~: 309158)
[PASS] testFuzz_OrderFieldManipulation() (gas: 2505788)
[PASS] testFuzz_OrderMatchingRaceCondition(uint256,uint8,uint8,uint8,bool) (runs: 260, μ: 2738741, ~: 2688117)
[PASS] testFuzz_WalletFactoryInteractions(uint256,bool,bool) (runs: 260, μ: 2476790, ~: 2507489)
[PASS] testGasOptimization() (gas: 3868080)
[PASS] testMatchERC1155toERC1155() (gas: 279044)
[PASS] testMatchMixedTokenTypes() (gas: 267406)
[PASS] testMatchOrdersAndMintBadSignature() (gas: 74829)
[PASS] testMatchOrdersAndMintCostCalculation() (gas: 258183)
[PASS] testMatchOrdersAndMintCoverage() (gas: 386714)
[PASS] testMatchOrdersAndMintEdgeCases() (gas: 399506)
[PASS] testMatchOrdersAndMintFilledError() (gas: 458678)
[PASS] testMatchOrdersAndMintFilledValidationBranches() (gas: 638830)
[PASS] testMatchOrdersAndMintMinCalculation() (gas: 506448)
[PASS] testMatchOrdersAndMintMinimumCalculationBranches() (gas: 506065)
[PASS] testMatchOrdersAndMintNonExistentWallets() (gas: 238674)
[PASS] testMatchOrdersAndMintRevokedSignature() (gas: 152732)
[PASS] testMatchOrdersAndMintWrongChainId() (gas: 77416)
[PASS] testMatchOrdersFeePaymentBranches() (gas: 804211)
[PASS] testMatchOrdersFeesComprehensive() (gas: 658788)
[PASS] testMatchOrdersOverfillProtection() (gas: 340475)
[PASS] testMatchOrdersOverfillValidationBranches() (gas: 389533)
[PASS] testMatchOrdersTokenPairMismatch() (gas: 186309)
[PASS] testMatchOrdersTokenPairMismatchBranches() (gas: 266056)
[PASS] testMatchOrdersWalletValidationBranches() (gas: 256713)
[PASS] testMatchOrdersWalletZeroAddress() (gas: 180079)
[PASS] testNonReentrantModifiers() (gas: 425278)
[PASS] testOnlyServerForAllFunctions() (gas: 91581)
[PASS] testOnlyServerModifier() (gas: 42701)
[PASS] testOnlyServerModifierBranches() (gas: 44650)
[PASS] testOnlyServerTxOriginCheck() (gas: 39586)
[PASS] testOrderCancellationEdgeCases() (gas: 126209)
[PASS] testPartialFillsWithBothTokenTypes() (gas: 620623)
[PASS] testPrecisionAndRounding() (gas: 394833)
[PASS] testSecurityEdgeCases() (gas: 404761)
[PASS] testSignatureEdgeCases() (gas: 72520)
[PASS] testSignatureVerificationAdditionalBranches() (gas: 101232)
[PASS] testSignatureVerificationCoverage() (gas: 405912)

[PASS] testTokenTransferCoverage() (gas: 2782961)
[PASS] testTransferTokenAdditionalBranches() (gas: 431908)
[PASS] testZeroFillAmounts() (gas: 196794)
Suite result: ok. 56 passed; 0 failed; 0 skipped; finished in 2.01s (13.26s CPU time)

**Ran 1 test suite in 2.02s (2.01s CPU time): 56 tests passed, 0 failed, 0 skipped (56 total tests)**

**Ran 10 tests for test/WalletFactory.t.sol:WalletFactory_Test**
[PASS] testFuzz_CreateWalletWithDifferentSalts(bytes32,bytes32,address) (runs: 263, μ: 339899, ~: 339899)
[PASS] testFuzz_GetWallet(address) (runs: 263, μ: 331369, ~: 331369)
[PASS] testFuzz_PredictWalletAddress(bytes32,address) (runs: 263, μ: 336503, ~: 336503)
[PASS] test_Constructor() (gas: 23409)
[PASS] test_CreateMultipleWallets() (gas: 636198)
[PASS] test_CreateWallet() (gas: 358274)
[PASS] test_GetWallet() (gas: 331818)
[PASS] test_PredictWalletAddress() (gas: 335886)
[PASS] test_RevertWhen_CREATE2Fails() (gas: 1040431029)
[PASS] test_RevertWhen_CreateWalletDuplicateOwner() (gas: 326246)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 197.16ms (547.59ms CPU time)

**Ran 9 tests for test/WalletBeacon.t.sol:WalletBeacon_Test**
[PASS] testFuzz_TransferOwnership(address) (runs: 263, μ: 51959, ~: 51959)
[PASS] testFuzz_UpgradeTo(address) (runs: 262, μ: 32420, ~: 32420)
[PASS] test_BeaconImplementationAddress() (gas: 11213)
[PASS] test_Constructor() (gas: 24991)
[PASS] test_Implementation() (gas: 18961)
[PASS] test_OwnershipTransfer() (gas: 50521)
[PASS] test_RenounceOwnership() (gas: 23137)
[PASS] test_RevertWhen_UpgradeToNonAdmin() (gas: 17027)
[PASS] test_UpgradeTo() (gas: 37468)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 302.01ms (171.27ms CPU time)

**Ran 23 tests for test/WalletImplementation.t.sol:WalletImplementation_Test**
[PASS] testFuzz_ExecuteBatchWithFuzzingSingleItem(uint256,uint256) (runs: 263, μ: 177016, ~: 178406)
[PASS] testFuzz_ExecuteTransactionWithFuzzing(uint256,uint256) (runs: 263, μ: 167078, ~: 168468)
[PASS] testFuzz_WithdrawWithDifferentAmounts(uint256) (runs: 263, μ: 159995, ~: 160664)
[PASS] test_ERC1155Reception() (gas: 121369)
[PASS] test_ERC1155TokenReceiver() (gas: 15109)
[PASS] test_ExecuteBatchSingleSig() (gas: 215949)
[PASS] test_ExecuteTransaction() (gas: 166891)

[PASS] test_ExecuteTransactionWithDifferentValues() (gas: 844625)
[PASS] test_Initialize() (gas: 214402)
[PASS] test_ReceiveEther() (gas: 16499)
[PASS] test_RevertWhen_ExecuteBatchArrayMismatch() (gas: 42951)
[PASS] test_RevertWhen_ExecuteBatchInvalidSignature() (gas: 52081)
[PASS] test_RevertWhen_ExecuteBatchNotServer() (gas: 31618)
[PASS] test_RevertWhen_ExecuteBatchTransactionFails() (gas: 1040435263)
[PASS] test_RevertWhen_ExecuteTransactionInvalidSignature() (gas: 40864)
[PASS] test_RevertWhen_ExecuteTransactionNotServer() (gas: 27562)
[PASS] test_RevertWhen_ReinitializeWallet() (gas: 14649)
[PASS] test_RevertWhen_TransactionFails() (gas: 1040433370)
[PASS] test_RevertWhen_WithdrawInsufficientBalance() (gas: 38926)
[PASS] test_RevertWhen_WithdrawNotServer() (gas: 27540)
[PASS] test_SupportsInterface() (gas: 13742)
[PASS] test_Withdraw() (gas: 146107)
[PASS] test_WithdrawWithCustomData() (gas: 111685)
Suite result: ok. 23 passed; 0 failed; 0 skipped; finished in 309.98ms (910.07ms CPU time)

**Ran 3 test suites in 313.91ms (809.15ms CPU time): 42 tests passed, 0 failed, 0 skipped (42 total tests)**

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

| FILE | % STMTS | % BRANCH | % FUNCS | % LINES |
|---|---|---|---|---|
| MarketFactory.sol | 100 | 100 | 100 | 100 |
| MarketImplementation.sol | 100 | 90 | 100 | 100 |
| Oracle.sol | 100 | 93.33 | 100 | 100 |
| OutcomeToken1155.sol | 100 | 100 | 100 | 100 |
| MatchingEngine.sol | 100 | 57.14 | 100 | 100 |
| WalletBeacon.sol | 100 | 100 | 100 | 100 |
| WalletFactory.sol | 100 | 100 | 100 | 100 |
| WalletImplementation.sol | 100 | 100 | 100 | 100 |
| **All Files** | **100** | **84.90** | **100** | **100** |

We are grateful for the opportunity to work with the Outcome team.

**The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.**

Zokyo Security recommends the Outcome team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo