HackenProof

SUMMARY REPORT

Ignite Market Audit Contest

SMART CONTRACTS CODE LANGUAGE

Solidity

REPORT CREATED

20.06.2025

AUDIT DURATION

06.05.2025 - 04.06.2025

PREPARED BY

HackenProof Team

HackenProof | Ignitemarket

HackenProof Crowdourced Audit is your solution to maximize the cybersecurity level with involvement of over 40k of the most skilled and experienced community researchers in the industry

Approved by	HackenProof team
Name	Ignite Market Audit Contest
Type	Smart Contracts
Technology	Solidity
Timeline	06.05.2025 - 04.06.2025
Deployed contract / GitHub	Public

HackenProof | Ignitemarket

Table of Contests

HackenProof | Ignitemarket

OVERVIEW

ABOUT AUDIT

BUDGET ALLOCATED	TOTAL REPORT SUBMITTED	SCOPE REVIEW COUNT
10 000	111	5274

ABOUT IGNITE MARKET

Ignite Market is a prection market based on gnosis conditional token and fixed product market maker on flare network leveraging their data connectors for providing real world data.

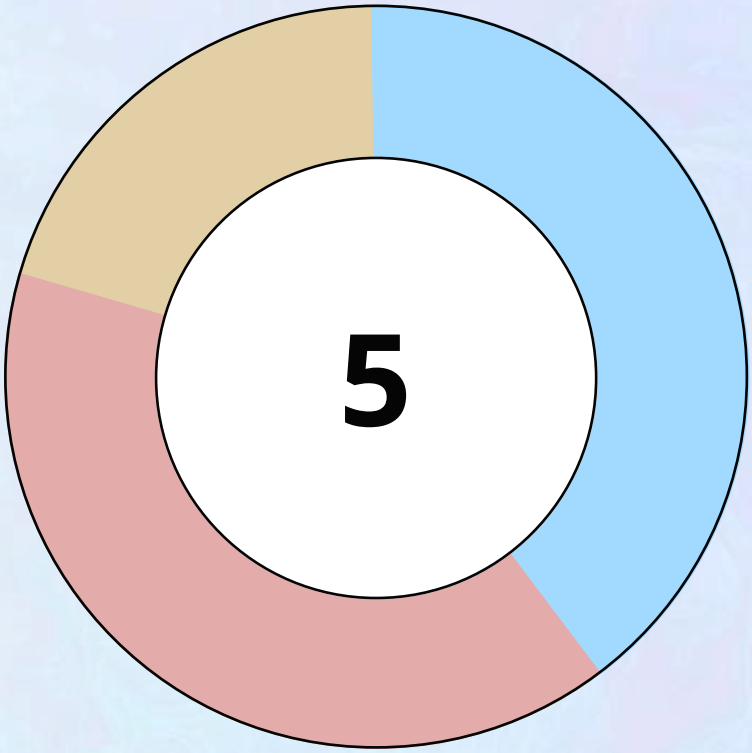
SCOPES AND TARGETS

Target	Type	Tech
https://github.com/hackenproof-public/ignite-market-smart-contracts	Smart contract	Solidity

HackenProof | Ignitemarket

FINDINGS

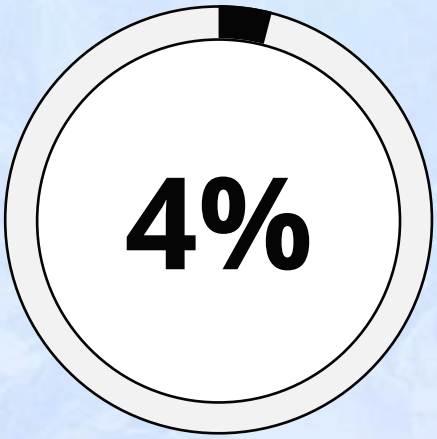
VALID REPORTS STATISTICS



- Critical - 0
- High - 2
- Medium - 1
- Low - 2
- Informative - 0

SIGNAL-TO-NOISE RATIO

Displays the ratio of the Valid reports and Received reports. Valid reports include Informative, Triaged, Paid, Resolved, Discosed statuses



FINDINGS LIST

Name	Severity	Submission date	State
IGNITEAC-106 Unrestricted ERC1155 Token Acceptance Leading to Pool Balance Manipulation	• High	03.06.2025	Disclosed

---

VULNERABILITY DETAILS

Summary

A donation attack is possible due to incorrectly implemented *onERC1155BatchReceived* and *onERC1155Received* functions, which can shift the weight/proportion assigned to each pool. All subsequent buys of that specific outcome/positionId will lead to buyers receiving less than they should (see PoC below). The original Gnosis FPMM contract (which this is a fork of) protects against this vulnerability by only accepting ERC115 transfers when operator is the FPMM contract address only, take a look

VULNERABILITY DETAILS

The vulnerability exists in the ERC1155 receiver implementation:

```
function onERC1155Received(
    address operator,
    address from,
    uint256 id,
    uint256 value,
    bytes calldata data
) external override returns(bytes4) {
    return this.onERC1155Received.selector;
}

function onERC1155BatchReceived(
    address operator,
    address from,
    uint256[] calldata ids,
    uint256[] calldata values,
    bytes calldata data
) external override returns(bytes4) {
    return this.onERC1155BatchReceived.selector;
}
```

Impact

1. Anyone can send ERC1155 tokens to the contract
2. This can artificially inflate pool balances
3. Affects calculations in *calcBuyAmount* and *calcSellAmount*
4. Can be used to manipulate market prices
5. May lead to incorrect fee calculations
6. Could be used to drain funds from the contract

Recommended Fix

1. Implement proper validation in receiver functions:

```
function onERC1155Received(
    address operator,
    address from,
    uint256 id,
    uint256 value,
    bytes calldata data
)
    external
    returns (bytes4)
{
    if (operator == address(this)) {
        return this.onERC1155Received.selector;
    }
    return 0x0;
}

function onERC1155BatchReceived(
    address operator,
    address from,
    uint256[] calldata ids,
    uint256[] calldata values,
    bytes calldata data
)
    external
    returns (bytes4)
{
    if (operator == address(this) && from == address(0)) {
        return this.onERC1155BatchReceived.selector;
    }
    return 0x0;
}
```

Tools Used

- Manual code review
- Hardhat

	IGNITEAC-37 [M-2] `FixedProductMarketMaker.initialize`: Gas Exhaustion from Unbounded Condition Complexity in `_recordCollectionIDsForAllConditions` Leads to Denial of Service for Market Creation	• Low	13.05.2025	Disclosed

VULNERABILITY DETAILS

Vulnerability Description

The *FixedProductMarketMakerFactory* contract allows any user to attempt the creation of a new *FixedProductMarketMaker* (FPMM) instance via its *createFixedProductMarketMaker* function. This function, in turn, calls the initialize function on the newly cloned FPMM contract.

The *FixedProductMarketMaker.initialize* function takes an array of *_conditionIds* as input. It then calls the internal function *_recordCollectionIDsForAllConditions* to set up the necessary state based on these conditions and their respective *outcomeSlotCounts* (retrieved from the *ConditionalTokens* contract). The *ConditionalTokens.prepareCondition* function, which is also permissionless, allows users to create conditions with up to 256 outcomes.

The critical vulnerability lies in the *_recordCollectionIDsForAllConditions* function. Its computational and gas complexity grows multiplicatively with the *outcomeSlotCount* of each condition and linearly with the number of conditions. Specifically, the number of *positionIds* to be generated and stored (*positionIds.push()*) is the product of all *outcomeSlotCounts*.

The *FixedProductMarketMaker.initialize* function lacks sufficient input validation to check:

1. The maximum number of conditions in the *_conditionIds* array.
2. The maximum *outcomeSlotCount* for any individual condition being used.
3. More importantly, the total combined complexity (e.g., the product of all *outcomeSlotCounts*, which determines the loop iterations and storage operations in *_recordCollectionIDsForAllConditions*).

A malicious actor (or even an unknowing user attempting to create a very complex market) can exploit this by:

1. Permissionlessly creating multiple conditions via *ConditionalTokens.prepareCondition*, each with a high *outcomeSlotCount* (e.g., 4 conditions with 70 outcomes each).
2. Calling *FixedProductMarketMakerFactory.createFixedProductMarketMaker* with this array of high-complexity condition IDs.

When the FPMM's initialize function executes *_recordCollectionIDsForAllConditions* with such input, the number of required *positionIds.push()* operations and recursive calls becomes enormous, leading to gas consumption that will far exceed the block gas limit. This causes the initialize transaction to revert due to an out-of-gas error.

The consequence is that while the FPMM contract instance (the clone) is deployed to a new address, its initialization fails. The market contract is thus "bricked" – it exists on-chain but is in an uninitialized and unusable state. This constitutes a Denial of Service for the creation of functional market instances if parameters leading to excessive complexity are provided. While the factory itself remains operational for valid parameters, this flaw allows for the creation of unusable market contracts, potentially leading to on-chain clutter. This also highlights a discrepancy with any documentation suggesting markets are solely curated by a "dedicated markets team," as the code does not enforce such curation or complexity limits at the point of creation.

Impact

- **Denial of Service for Specific Market Creation**: Any attempt to create a FixedProductMarketMaker with a combination of conditions and outcome counts that exceeds a practical gas limit will fail. This prevents such markets from being successfully initialized and becoming operational.

- **Bricked Contract Instances:** When the initialization transaction reverts, the cloned FixedProductMarketMaker contract address still exists on-chain but in an uninitialized and unusable state. This can lead to "dead" contract deployments.
- **Resource Consumption:** Attackers can intentionally trigger these failed deployments, consuming their own gas but also potentially causing confusion or a perception of unreliability if many such bricked contracts appear to be associated with the factory.
- **Mismatch with Documented Curation:** If the protocol's documentation suggests that markets are curated by a dedicated team, this vulnerability demonstrates that the smart contract code does not enforce such curation or complexity limits at the point of creation, allowing anyone to trigger this DoS.

Recommended Mitigation

1. Implement Strict Input Validation in *FixedProductMarketMaker.initialize*:

- **Limit the Number of Conditions:** Enforce a maximum length for the *_conditionIds* array (e.g., 3-5 conditions).
- **Limit *outcomeSlotCount* per Condition**: While *ConditionalTokens* allows up to 256, the FPMM could enforce a lower practical limit for conditions it uses (e.g., 50-100 outcomes per condition).
- **Calculate and Limit Total Complexity:** Before calling *_recordCollectionIDsForAllConditions*, iterate through the provided *_conditionIds*, fetch their *outcomeSlotCounts* from the *ConditionalTokens* contract, and calculate the product of these counts. If this product (which represents the total number of *positionIds* to be generated) exceeds a predetermined safe maximum (e.g., a few thousand, based on gas testing), the initialize function should revert with a clear error message. This is the most robust check.

```
// Example check within initialize()
uint totalPositionComplexity = 1;
if (_conditionIds.length > MAX_CONDITIONS_ALLOWED) {
    revert TooManyConditions();
}
for (uint i = 0; i < _conditionIds.length; i++) {
    uint outcomeCount = conditionalTokens.getOutcomeSlotCount(_conditionIds[i]);
    if (outcomeCount == 0 || outcomeCount > MAX_OUTCOMES_PER_CONDITION_ALLOWED) { // also check if condition is prepared
        revert InvalidOutcomeCountForCondition();
    }
    if (MAX_TOTAL_POSITION_COMPLEXITY / outcomeCount < totalPositionComplexity) { // Check for overflow before multiplication
        revert TotalComplexityExceedsLimit();
    }
    totalPositionComplexity *= outcomeCount;
}
if (totalPositionComplexity > MAX_TOTAL_POSITION_COMPLEXITY) {
    revert TotalComplexityExceedsLimit();
}
// Proceed with _recordCollectionIDsForAllConditions
```

2. Access Control for Market Creation (Consider aligning with documentation):

- If the intent is truly for markets to be curated by a "dedicated markets team" as per documentation, consider adding access control to the *FixedProductMarketMakerFactory.createFixedProductMarketMaker* function. This would restrict market creation privileges to whitelisted addresses. This is a larger design decision but would directly address the mismatch between documentation and code behavior. However, the input validation (Point 1) should be implemented regardless of access control to prevent even privileged users from accidentally bricking markets.

Name	Severity	Submission date	State
IGNITEAC-32 Users can skip calling FixedProductMarketMaker.buy() and directly call ConditionalTokens.splitPosition() to get more conditional tokens than intended	• High	13.05.2025	Disclosed

VULNERABILITY DETAILS

When users call *buy()* in FixedProductMarketMaker.sol, they trade their collateral amount for a portion of the conditional token. The exchange rate is calculated through the AMM. Note that they only get 1 type of conditional token regardless of the number of outcomes available since they are only purchasing that particular outcome.

For example, for a condition with 3 outcomes, the user buys the 3rd outcome which he thinks is the most probable (index 2).

```
function buy(uint investmentAmount, uint outcomeIndex, uint minOutcomeTokensToBuy) external nonReentrant() {
        require(canTrade(), "trading not allowed");
        require((investmentAmount * 100) / fundingAmountTotal <= 10, "amount can be up to 10% of fundingAmountTotal");

        uint outcomeTokensToBuy = calcBuyAmount(investmentAmount, outcomeIndex);
        require(outcomeTokensToBuy >= minOutcomeTokensToBuy, "minimum buy amount not reached");

>       collateralToken.safeTransferFrom(msg.sender, address(this), investmentAmount);
        uint feeAmount = (investmentAmount * fee) / ONE;
        feePoolWeight += feeAmount;
        uint investmentAmountMinusFees = investmentAmount - feeAmount;

        collateralToken.forceApprove(address(conditionalTokens), investmentAmountMinusFees);
        splitPositionThroughAllConditions(investmentAmountMinusFees);

>       conditionalTokens.safeTransferFrom(address(this), msg.sender, positionIds[outcomeIndex], outcomeTokensToBuy, "");
        emit FPMMBuy(msg.sender, investmentAmount, feeAmount, outcomeIndex, outcomeTokensToBuy);
    }
```

Instead of getting one type of conditional token, the user can directly call *splitPosition()* and fill in all the parameters (the parameters are known with no access control to the function), this way the user can get all the different types of conditional tokens with just one amount.

```
  function splitPosition(
        IERC20 collateralToken,
        bytes32 parentCollectionId,
        bytes32 conditionId,
        uint[] calldata partition,
        uint amount
    ) external {
        require(partition.length > 1, "got empty or singleton partition");
        uint outcomeSlotCount = payoutNumerators[conditionId].length;
        require(outcomeSlotCount > 0, "condition not prepared yet");

        uint fullIndexSet = (1 << outcomeSlotCount) - 1;
        uint freeIndexSet = fullIndexSet;
        uint[] memory positionIds = new uint[](partition.length);
        uint[] memory amounts = new uint[](partition.length);
        for (uint i = 0; i < partition.length; i++) {
            uint indexSet = partition[i];
            require(indexSet > 0 && indexSet < fullIndexSet, "got invalid index set");
            require((indexSet & freeIndexSet) == indexSet, "partition not disjoint");
            freeIndexSet ^= indexSet;
            positionIds[i] = CTHelpers.getPositionId(collateralToken, CTHelpers.getCollectionId(parentCollectionId, conditionId,
>           amounts[i] = amount;
        }

        if (freeIndexSet == 0) {
            if (parentCollectionId == bytes32(0)) {
                require(collateralToken.transferFrom(msg.sender, address(this), amount), "could not receive collateral tokens");
            } else {
                _burn(
                    msg.sender,
                    CTHelpers.getPositionId(collateralToken, parentCollectionId),
                    amount
                );
            }
        } else {
            _burn(
                msg.sender,
                CTHelpers.getPositionId(collateralToken,
                    CTHelpers.getCollectionId(parentCollectionId, conditionId, fullIndexSet ^ freeIndexSet)),
                amount
            );
        }

>       _mintBatch(
            msg.sender,
            positionIds,
            amounts,
            ""
        );
        emit PositionSplit(msg.sender, collateralToken, parentCollectionId, conditionId, partition, amount);
    }
```

The function will mint conditional tokens for all the *positionIds*.

Name	Severity	Submission date	State
IGNITEAC-14 Initial voters are able to renounce their role without reduction of the voter count	• Low	09.05.2025	Disclosed

---

VULNERABILITY DETAILS

The *IgniteOracle.sol*, responsible for question initialization and finalization, relies on a set of voters for non-automatic question resolutions. The project will initially work with a set of voters before transitioning to a governance DAO model, based on token holdings. To achieve this, a VOTER role is created and granted and each granting increments the number of voters and each revoking of a VOTER decreases the number of voters. This is achieved via implementing Openzeppelin's *AccessControl.sol* and overriding it's methods. However the method *renounceRole()* is left completely public:

```
function renounceRole(bytes32 role, address callerConfirmation) public virtual {
    if (callerConfirmation != _msgSender()) {
        revert AccessControlBadConfirmation();
    }

    _revokeRole(role, callerConfirmation);
}
```

This would allow any VOTER to leave the system without decrementing the voter count, as it should, leaving the contract in an incorrect state. On it's own, a voter is not incentivized to do this. However, in the scenario where voters are removed, but re-added at a later point, a voter can front-run the admin's call to revoke and renounce it before him. If at a later point, this voter gets re-added, there would be 1 more voter in the total that there are voters available, potentially impacting question resolutions when the vote is close to the quorum percentage.

Name	Severity	Submission date	State
IGNITEAC-12 Question creation can be repeatedly reverted by front-running the id	• Medium	09.05.2025	Disclosed

---

VULNERABILITY DETAILS

The *IgniteOracle.sol* contract is the main entry-point for creation of questions and their finalizations and voting. It does a bunch of necessary sanity checks for the provided question parameters and the protocol specific API sources, after which it invokes the *ConditionalTokens.sol*'s *prepareCondition()*, providing the condition id, it's own address and the outcome slots. This is mostly forked logic, however it contains a front-running risk, due to:

1. *prepareCondition()*'s permissionless availability
2. Flare network's public mempool/pending transactions explorer

The core of the vulnerability is point 1, since looking the *prepareCondition()*:

```
function prepareCondition(address oracle, bytes32 questionId, uint outcomeSlotCount) external {
    require(outcomeSlotCount <= 256, "too many outcome slots");
    require(outcomeSlotCount > 1, "there should be more than one outcome slot");
    bytes32 conditionId = CTHelpers.getConditionId(oracle, questionId, outcomeSlotCount);
    require(payoutNumerators[conditionId].length == 0, "condition already prepared");
    payoutNumerators[conditionId] = new uint[](outcomeSlotCount);
    emit ConditionPreparation(conditionId, oracle, questionId, outcomeSlotCount);
}
```

It does not do any check on the caller, as long as we provide valid parameters. However, the function itself makes sure that the provided question id has not been yet prepared. This opens up the possible path:

1. The admin attempts to initialize a question and invokes *initializeQuestion()* with the correct parameters and an arbitrary question id
2. The transaction occupies Flare's pending list, which any actor can observe
3. A malicious actor sees the transaction and front-runs it via directly calling *ConditionalTokens#prepareCondition()* and passing the oracle address, question id and outcome slots
4. The admin's transaction will fail due to the condition being already prepared

Since there is no question behind the arbitrarily prepared condition, it cannot be finalized nor voted for either, resulting in a DOS of the initialization for no cost, except for the gas it takes to front-run, which is fairly low on Flare

An article on the same vulnerability, found on the famous platform Polymarket, by Trust Security, can be found here: https://www.trust-security.xyz/post/no-more-bets

HackenProof | Ignitemarket

SECURITY RESEARCHERS

TOP HACKERS

Researcher	Total findings	Place
@hashbrown	5	1
 @0xTonraq	4	2
@iam0ti	1	3
@laarobi	2	4
 @sensei	1	5

CONCLUSION

ABOUT IGNITE MARKET

The audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract.

It is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.

TECHNICAL DISCLAIMER

Smart contracts are deployed and executed on blockchain platform. The platform, its programming language, and other software related to the smart contract can have its vulnerabilities that can lead to hacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.