Phage SECURITY

Foresight

Security Overhaul

Conducted by:

Pyro, Security Researcher
Deth, Security Researcher
YanecaB, Junior Security Researcher

15.08.2025

Table of Contents

Disclaimer

This report represents a high-level security review, not a comprehensive audit. Due to time constraints unsuitable for conducting a proper security audit, the review was limited in scope and depth. Security researchers employed manual review techniques to identify basic vulnerabilities and provide recommendations to improve the overall security posture of the smart contracts.

This time-constrained security review cannot guarantee the absence of vulnerabilities and should not be considered equivalent to a full security audit. While efforts were made to identify fundamental security issues and enhance the project's security baseline, the limited timeframe may have prevented the discovery of more complex or subtle vulnerabilities that would typically be uncovered in a comprehensive audit process.

System overview

Foresight is a prediction market built on Katana, utilizing AUSD as collateral for trading conditional tokens representing different event outcomes. The system employs LMSR (Logarithmic Market Scoring Rule) automated market maker to provide liquidity and dynamic pricing, while the ConditionalTokens contract manages ERC1155-based outcome positions that can be minted, traded, and redeemed upon condition resolution.

Executive summary

Overview

Project Name	Foresight
Repository	Foresight
Commit hash	011e672b709198dd14e39eb7eac81284c1557ad0
Remediation	735132354600c2b236ba9a7879cec6c1f5bfa864
Methods	Manual review

Timeline

Audit kick-off	05.08.2025
End of audit	09.08.2025
Remediations start	12.08.2025
Remediations end	15.08.2025

Scope

src/lmsrMarket/MarketMaker.sol

src/lmsrMarket/LSLMSRMarketMaker.sol

src/lmsrMarket/LSLMSRMarketMakerFactory.sol

Issues Found

Severity	Count
High	3
Medium	2
Low	2

Findings

High Severity

[H-01] `ammMint` messes up all collateral backing

The original Gnosis CTF contract uses `splitPosition` for creating new positions, where it would mint all outcomes. This will maintain our collateral proportion. For example, if we pay 100 USDC, it would mint us 100 YES and 100 NO tokens. Later in the call, if the user wants only YES, we can sell the NO and get back some more YES.

Our contracts operate differently. `trade` only mints the desired tokens and ignores the rest:

```
    if (outcomeTokenNetCost > 0) {
        collateralToken.safeTransferFrom(msg.sender, address(this),
            uint256(netCost));
        require(collateralToken.approve(address(pmSystem), uint256(
            outcomeTokenNetCost)));
        uint256[] memory amounts = new uint256[](atomicOutcomeSlotCount);
        for (uint256 i = 0; i < atomicOutcomeSlotCount; i++) {
            // [YES = 50e6, NO = 0]
            amounts[i] = uint256(outcomeTokenAmounts[i]);
        }
        pmSystem.ammMint(collateralToken, uint256(outcomeTokenNetCost),
            address(this), positionIds, amounts);
        // splitPositionThroughAllConditions(uint(outcomeTokenNetCost));
    }
```

The impact is best seen with the example below:

```
Initial state after funding:

1. Market receives: 1000 USDT collateral
- splitPositionThroughAllConditions(1000) creates:
  - 1000 YES tokens
  - 1000 NO tokens
Q_total = 2000, b = (alpha * 2000) / 1e18

2. User buys 500 more YES tokens:
- ammMint() creates: 500 YES tokens (ONLY)
  - 1500 YES tokens
  - 1000 NO tokens
```

Now when we calculate `Q_total` and `b` and check our collateral backing, we get a different ratio from before:

```
Q_total = 1500 YES + 1000 NO = 2500
b = (alpha * 2500) / 1e18
Collateral backing = 1000 + 500 = 1500 USDT
```

The math will assume 2500 USDT in backing, which would mess up internal accounting, lead to constantly changing prices and potential for MEV.

Recommendation

Given the current design, the best suggestion would be to implement the old Gnosis CTF functions and mint on both sides.

[H-02] Any user can drain the referral contract

Anyone can register a referral code and can set `marketAddress` to their smart contract address:

```
function registerReferralCode(address user, bytes32 conditionId,
   uint256 outcomeIndex, address marketAddress)
   public
   returns (string memory referralCode)
{
   // keccak256(abi.encodePacked(user, conditionId, outcomeIndex));
   referralCode = generateReferralCode(user, conditionId, outcomeIndex);
   // Check if this referral code already exists
   require(referralCodes[referralCode].user == address(0),
      "Referral code already exists");
   // Register the referral code
   referralCodes[referralCode] = ReferralCode({
      user: user,
      conditionId: conditionId,
      outcomeIndex: outcomeIndex,
      marketAddress: marketAddress
   });
   emit ReferralCodeGenerated(user, referralCode, conditionId,
      outcomeIndex);
   return referralCode;
}
```

This will make `processReferral` pass without an issue if called by their contract as their contract will be the `marketAddress == refCode.marketAddress` and also the `msg.sender - msg.sender == marketAddress`.

```
function processReferral(address referee, string memory referralCode,
   uint256 earnings, address marketAddress)
   external
{
   ReferralCode memory refCode = referralCodes[referralCode];
   require(refCode.user ≠ address(0), "Invalid referral code");
   require(refCode.user ≠ referee, "Cannot refer yourself");
   require(earnings > 0, "Earnings must be greater than zero");
   require(marketAddress == refCode.marketAddress, "Market address
      mismatch");
   // Only allow calls from the associated market maker
   require(msg.sender == marketAddress, "Only market maker can call");
```

Then the function will continue to `_convertUSDToShares` where we would grant them approval and call trade, which in turn can be just a `transferFrom` from the referral contract to our malicious contract.

```
   require(collateralToken.approve(address(market), costUint),
      "Failed to approve market for collateral");
   // Execute the trade with fee-free parameters
   int256 actualCost = market.trade(
      outcomeTokenAmounts,
      int256(costUint), // collateralLimit - use exact calculated cost
      "", // empty referralCode to avoid recursion
      true // isReferralContract = true for fee-free trade
   );
```

This would allow anyone to drain the referral system.

The POC for this is in `test/PhageSecurity/PhageSecurity.t.sol` - `test_02_ReferralSystemExploit`

Recommendation

Whitelist specific market addresses in order to make sure only real markets can be used. Also consider adding access control so that regular users cannot refer to another address controlled by them steal part of the fees.

[H-03] Unauthorized redemption in `redeemPositionForUser`

The `ConditionalTokens::redeemPositionsForUser` function lacks access control, allowing anyone to redeem positions on behalf of any user.

I as a malicious user can exploit this by redeeming another user's positions without consent. And setting `donationPercentages` to `100%` and redirecting the full payout to myself. => This effectively allows front-running or griefing attacks where users lose their collateral.

Recommendation

Implement access control for `redeemPositionsForUser`, e.g. an approval mechanism, similar to how ERC-20/721 approvals work.

```
// Mapping of user => operator => approved
mapping(address => mapping(address => bool)) private redemptionApprovals;
```

```
/// @notice Approves or revokes a third-party to redeem positions on your behalf
function setRedemptionApproval(address operator, bool approved) external {
   redemptionApprovals[msg.sender][operator] = approved;
}

/// @notice Returns whether an operator is approved to redeem for a user
function isRedemptionApproved(address account, address operator) public view
   returns (bool) {
   return redemptionApprovals[account][operator];
}
```

And update the start of the `redeemPositionsForUser` to be like:

```
   require(
      account == msg.sender || isRedemptionApproved(account, msg.sender),
      "Not authorized to redeem for this user"
   );
```

If you want you can also implement donation restriction, so that the approver does not donate 100% of the tokens to himself.

Medium Severity

[M-01] Mixed buy/sell trades can brick the market

When trading, a user can specify if they want to buy/sell any atomic position they want. This is fed to `calcNetCost`, which tells if after they buy/sell their chosen outcome tokens, they have to pay or receive collateral.

If the user is just buying or just selling, it's trivial as they always have to pay/receive, but if they buy and sell in the same transaction, then things get more complicated.

Let's examine where the `outcomeTokenNetCost` is handled (if user pays/receives):

```
// Pays
 // Handle collateral transfer and position splitting for buys
      if(outcomeTokenNetCost > 0) {
         collateralToken.safeTransferFrom(msg.sender, address(this), uint(
            netCost));
         require(collateralToken.approve(address(pmSystem), uint(
            outcomeTokenNetCost)));
         // mint shares based on token amount
         uint[] memory amounts = new uint[](atomicOutcomeSlotCount);
         for (uint i = 0; i < atomicOutcomeSlotCount; i++) {

            amounts[i] = uint(outcomeTokenAmounts[i]);
         }
         pmSystem.ammMint(collateralToken, uint(outcomeTokenNetCost),
            address(this), positionIds, amounts);
         // splitPositionThroughAllConditions(uint(outcomeTokenNetCost));
      }

// Receives
if(outcomeTokenNetCost < 0) {
       uint[] memory amounts = new uint[](atomicOutcomeSlotCount);
       for (uint i = 0; i < atomicOutcomeSlotCount; i++) {
          amounts[i] = uint(-outcomeTokenAmounts[i]);
       }

       pmSystem.ammBurn(collateralToken, uint(-netCost) + tradingFeeAmount ,
          positionIds, amounts);
         // mergePositionsThroughAllConditions(uint(-outcomeTokenNetCost));
      }
```

Notice the loop. We'll examine the paying route, as it's hit first:

```
   for (uint i = 0; i < atomicOutcomeSlotCount; i++) {
      amounts[i] = uint(outcomeTokenAmounts[i]);
   }
```

The code always casts `outcomeTokenAmounts[i]` to `uint`, which is fine if the user is only buying, thus `outcomeTokenAmounts > 0`, but if they are selling then it's `< 0`. Casting a negative integer to `uint` causes the value to underflow into a giant uint.

Example, where we try to sell `50e6` of `index0` and buy `75e6` of `index1`:

```
outcomeTokenAmounts: 115792089237316195423570985008687907853269984665640564039
   457584007913079639936
outcomeTokenAmounts: 75000000
```

The same thing happens when receiving: it treats each `outcomeTokenAmounts` as a sell. If it was a buy, then `-outcomeTokenAmounts[i]` turns it negative.

In its normal state, the code reverts due to an overflow when minting, as we try to mint more

than the max possible tokens. However, given the right market and right conditions, it might mess up the whole internal accounting and inflate one of the tokens.

POC is in `test/PhageSecurity.t.sol` - `test_01_BuyAndSellSameTX`

Recommendation

In general, it's dangerous to have buys and sells in the same transaction. The code below can be used to confirm that all `outcomeTokenAmounts` point in the same direction (or are `0`):

```
function trade( ... ) {
   // Validate that all outcomeTokenAmounts are in the same direction (
   // all positive/zero or all negative/zero)
   bool hasPositive = false;
   bool hasNegative = false;

   for (uint i = 0; i < outcomeTokenAmounts.length; i++) {
      if (outcomeTokenAmounts[i] > 0) {
         hasPositive = true;
      } else if (outcomeTokenAmounts[i] < 0) {
         hasNegative = true;
      }

      // Require that we don't have both positive and negative amounts
      require(!(hasPositive && hasNegative), "Mixed buy and sell orders
         not allowed in single trade");
   }
```

Low Severity

[L-01] Donation not applied when minting ERC1155 positions

The `redeemPositionsForUser` function includes a donation fee mechanism meant to reward the caller (`msg.sender`). The actual transfer logic is handled by `_handleTransfers`. When `context.parentCollectionId == bytes32(0)`, the contract transfers `totalPayout - totalDonationAmount` to the `account` and `totalDonationAmount` to `msg.sender`, as expected. However, when `context.parentCollectionId ≠ bytes32(0)`, the function mints ERC1155 positions instead of transferring ERC20 tokens — but it incorrectly mints the full `totalPayout` to the `account`, and `msg.sender` receives nothing. This breaks the donation logic and deprives the caller of their intended fee.

```
function _handleTransfers(
   RedeemContext memory context,
   address account,
   uint totalPayout,
   uint totalDonationAmount
) internal {
   if (context.parentCollectionId == bytes32(0)) {
      require(
         context.collateralToken.transfer(account, totalPayout -
            totalDonationAmount),
         "could not transfer payout to account"
      );
      require(
         context.collateralToken.transfer(msg.sender,
            totalDonationAmount),
         "could not transfer donation"
      );
   } else {
      _mint(
         account,
         CTHelpers.getPositionId(context.collateralToken,
            context.parentCollectionId),
         totalPayout,
         ""
      ); //@audit the donation payment for the `msg.sender` is skipped
         and the account profits it
   }
}
```

Recommendation

Update the `else` block to be something like:

```
} else {
 _mint(
    account,
    CTHelpers.getPositionId(context.collateralToken,
       context.parentCollectionId),
    totalPayout - totalDonationAmount,
    ""
 );
 _mint(
    msg.sender,
    CTHelpers.getPositionId(context.collateralToken,
       context.parentCollectionId),
    totalDonationAmount,
    ""
 );
}
```

[L-02] Using regular `approve` will break the contracts if USDT is used as collateral

In a few places around the code `approve` is used. Currently that is not a problem as the main collateral is planed to be AUSD, however in the future the market may be switched to USDC or USDT. Due to the contract using approve USDT will not work correctly and will cause some part of the code to be unusable.

```
if(outcomeTokenNetCost > 0) {
   collateralToken.safeTransferFrom(msg.sender, address(this), uint(
      netCost));

   require(collateralToken.approve(address(pmSystem), uint(
      outcomeTokenNetCost)));
```

Note that in `ConditionalTokens` regular transfer is also used, which again would not work with USDT as the transfer can revert, but it will still return true.

```
require(
   collateralToken.transferFrom(
      msg.sender,
      address(this),
      collateralAmount
   ),
   "could not receive collateral tokens"
);
```

Recommendation

Use `safeIncreaseAllowance` instead of `approve` and `safeTransfer` instead of the require transfer method.