TopCut Finance Security Review

Security Researcher: Carrotsmuggler

June 5, 2025 - June 10, 2025

SUMMARY

Protocol Summary

Protocol Name	TopCut Finance
Repository	https://github.com/PossumLabsCrypto/TopCut/tree/ce88e78ebd782fc88399f9122a4e66febadf7f9d
Date	June 05 2025 - June 10 2025
Protocol Type	Prediction markets
Scope	ce88e78ebd782fc88399f9122a4e66febadf7f9d
Security Researcher	carrotsmuggler

Findings Summary

Severity	Total	Fixed	Open	Acknowledged
High	-	-	-	-
Medium	5	3	-	2
Low	2	1	-	1
Info	9	2	-	7

Introduction

A time-boxed security review was conducted on the linked commit. Fixes were reviewed and merged with the main branch at commit *7cd27f959552cc61efff4cc40313d5b178c6ae6c.*

About TopCut Finance

The TopCut finance protocol implements a prediction market that allows users to predict the price of an asset. After the duration of the trade, further trades are locked until settlement. At settlement, users with predictions closest to the actual oracle price are rewarded 10x their initial investment. There is 1 winner for every 11 participants. Part of the funds are redirected to the vault contract, where affiliates can take their share of the revenue.

About Carrotsmuggler

Carrotsmuggler is an independent smart contract security researcher. He serves as an associate security researcher at Spearbit, conducted multiple security reviews as a part of the Pashov audit group, and has 10+ top 3 standings on various security platforms such as Code4rena and Sherlock.

Scope

The following contracts were in scope:

- TopCutMarket.sol
- TopCutNFT.sol
- TopCutVault.sol

Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

[M-01] Trade can be assigned to unexpected cohort

Severity
Impact: High
Likelihood: Low

Description
When a user submits a trade, they have no control over which cohort they are submitting to. While this is fine since it will be handled by the UI, there can be odd behaviour for trades placed around *settlementTime*.

```
function castPrediction(address _frontend, uint256 _refID, uint256 _price)
external payable {
```

Currently, before *settlementTime*, trades are assigned to cohort A. After *settlementTime*, anyone can call the settle function and, provided an updated oracle data is available, the active cohort will flip from A to B.

If a user submits a transaction around this time, they might have intended their transaction to be executed in cohort A. However, due to delays on the RPC URL or high traffic on the chain, their transaction can get delayed for a few seconds, and a *settle* call might slip through in the meantime.

Now, users who were aiming for cohort A with a known payout ratio and known price predictions from other users will suddenly be assigned to cohort B, where they lose their information advantage.

Mitigation recommendation
Consider allowing an *activeCohort* parameter or a *deadline* parameter to the prediction function. This way, delayed transactions can get cancelled if they are too late or if the *activeCohort* passed in does not match the current state of the contract

Remediation
Fixed by adding an *activeCohort* parameter.

[M-02] Missing chainlink checks

Severity
Impact: High
Likelihood: Low

Description

The contract lacks 2 critical checks required when using Chainlink price feeds:

1. Sequencer uptime check for L2 deployments
2. Price staleness check

When using chainlink oracles on L2 networks such as Arbitrum, it is advised to check the uptime state of the sequencer.

Chainlink provides this functionality, and a thorough description can be found here: https://docs.chain.link/data-feeds/l2-sequencer-feeds

In brief, if a sequencer goes down, Oracle updates will stop. Users can directly interact with the L1 Arbitrum validator contract and still submit transactions, which are then held in a special buffer. Once the sequencer turns back on, these buffered transactions gain priority. In such a case, users can get an unfair advantage if their transaction gets accepted before the actual oracle update transaction.

While the contract makes sure that the *updatedAt* is fresher than the *settlementTime*, if there are delays in calling this function, the oracle can still end up using stale values of the oracle

Mitigation recommendation

1. In the *settleCohort* function, implement a sequencer uptime check. A reference implementation can be found here: https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-consumer-contract

2. Implement a staleness check, where the chainlink oracle data returned is expected to have been updated within the last 2-3 hours (for ETH/USD) feeds. Each oracle has a different heartbeat, and a staleness check of 2-3x the heartbeat value is generally sufficient.

Remediation
Fixed following recommendations.

[M-03] Affiliates can make profitable donations

Severity
Impact: High
Likelihood: Low

Description

The *TopCutVault updatePoints* function is an open function, meaning anyone can make direct calls to it, donating ETH and increasing their share of points. The issue is that for affiliates, there are certain scenarios where they can take out more ETH than they donate, making it a profitable heist.

Firstly, anyone can become an affiliate by minting a *TopCutNFT* for themselves.

When someone calls *updatePoints* directly on the contract, they pass in a *msg.value*. Their associated *affiliatePoints[_refID]* also increase by the same amount. For this to be a profitable donation, the extra rewards they will get due to this increased amount of points must be higher than their initial point investment.

After working out the maths, it was found that profitable donations can be made as long as,

$$B > TR + P$$

where B is the eth balance in the contract, TR is the *totalRedeemedAP* and P is the points of the manipulator.

Consider the following scenario:

Vault has 1.5e18 eth (B). No one has redeemed, so *totalRedeemedAP* is 1e18 (TR) and the manipulator has no points (P=0). This is a profitable situation, since

$$B > TR + P$$

The manipulator can then call updatePoints with *msg.value* of 1e18. Then the contract balance changes to 2.5e18, and the manipulator's points change to 1e18. Now, when they call quoteAffiliateReward, they are entitled to,

$$2.5e18 * \frac{1e18}{1e18 + 1e18} = 1.25e18$$

So they invested 1e18 and got back 1.25e18 without providing any services.

```
function quoteAffiliateReward(uint256 _pointsRedeemed) public view returns
```

```
(uint256 ethReward) {
     uint256 ethBalance = address(this).balance;
     uint256 newTotalAffiliatePoints = totalRedeemedAP + _pointsRedeemed;

     ///@dev Calculate the ETH rewards received by the affiliate after pool
protection (slippage as in AMM)
     ethReward = (ethBalance * _pointsRedeemed) / newTotalAffiliatePoints;
 }
```

Thus, due to the open updatePoints function, they were able to make a profit. If updatePoints was closed, this would not have been possible, since only 5% of their donation would go to the vault.

Mitigation recommendation

Consider restricting the addresses that can call *updatePoints*. It can be only the market, or a whitelist of addresses representing different markets.

Remediation

Acknowledged and accepted as a design decision. Affiliates are expected to regularly redeem AP, making the scenario unlikely in actual operation.

[M-04] Incorrect *totalRedeemedAP* update

Severity

Impact: Medium
Likelihood: Medium

Description

The *totalRedeemedAP* value is always updated with *affiliatePoints[_refID]* even though only the passed-in *_pointsRedeemed* amount is utilized.

```
totalRedeemedAP = (totalRedeemedAP + affiliatePoints[_refID] <
MAX_AP_REDEEMED)
     ? totalRedeemedAP + affiliatePoints[_refID]
     : MAX_AP_REDEEMED;
```

Due to this, *totalRedeemedAP* will increase faster than intended due to the incorrect parameter.

Mitigation recommendation

Change *affiliatePoints[_refID]* to *_pointsRedeemed*.

Remediation

Fixed following recommendation.

[M-05] Protocol can lose if there are less than 11 participants in any cohort

Severity

Impact: High
Likelihood: Low

Description

The system works by rewarding 10x to every 1 out of 11 participants.

So for every 11 units of funds received,

The winner gets 10 units
vault gets 5% = 0.55 units
frontend gets 3% = 0.33 units
keeper gets 1% = 0.11 units

The remaining 0.01 units remain with he vault.

However, due to how the winner choice system is implemented, the payouts still happen the same way if there are fewer than 11 participants, netting a loss of funds for the protocol, which can result in insolvency.

```
cohortWinners = (_cohortSize > 11) ? _cohortSize / 11 : 1; // Minimum 1
winner
```

As seen above, a minimum of 1 winner is chosen even if there isn't enough participation.

This situation will always lead to a loss to the protocol, and there are no mitigations implemented at the smart contract level.

Mitigation recommendation

The team has communicated that this will be mitigated on an operational level. However, this is still a threat, since cohorts hidden in the UI can still be participated in by calling the smart contracts directly. This would require constant monitoring in order not to suffer material losses.

A better mitigation would be to cancel cohorts with not enough participation and allow users to claim back their funds minus the fees.

Remediation

Acknowledged as an accepted operational risk.

[L-01] Undefined behaviour at *block.timestamp == settlementTime*

Description

The *castPrediction* function implements a time stamp check to let trades go through.

```
if (block.timestamp > settlementTime) revert WaitingToSettle();
```

The *settleCohort* function also implements a time-stamp check for settlement.

```
if (updatedAt < settlementTime) revert StaleOraclePrice();
```

The issue is that at *block.timestamp == settlementTime*, both these functions are callable. Thus, the behaviour of the system becomes unpredictable, and whichever transaction comes first gets executed. This can lead to unexpected results, as two transactions in the same block can have different outcomes depending on which gets executed first.

Mitigation recommendation

Consider changing one of the time checks to a <= or >=. That way, even at *block.timestamp == settlementTime*, only one option is valid.

Remediation

Fixed by changing the check to a >= in the prediction function.

[L-02] Cohorts can be open for less than *TRADE_DURATION* time

Description

The protocol is designed such that each cohort is open for *TRADE_DURATION* time. After this, the cohort is locked and eventually settled. During settlement, the *nextSettlement* is set to the current *settlementTime* + *TRADE_DURATION*. This means that if there was a delay in calling the settlement, the next cohort will lose that delay amount of time.

```
nextSettlement = settlementTime + TRADE_DURATION;
```

Say the trade duration is 24 hours. *settlementTime* is at h (hour) =0. Say there was a delay and the settlement was called at h=6 hours. Then *settlementTime* will be reset to h=24. So the new cohort will be open from h=6 to h=24, so for 18 hours only.

This same issue is also present in the *_distributeLoyaltyReward* function of the TopCutVault contract. Here, if the reward distribution is late, the next round of rewards will be distributed after a time less than *DISTRIBUTION_INTERVAL* seconds since the last one.

Mitigation recommendation

Consider changing the update to be based on *block.timestamp* instead of the last *settlementTime*.

Remediation

Acknowledged as a design decision.

[I-01] Conflicting distribution interval

The *TopCutVault.sol* contract has a comment saying that a minimum wait of 2 weeks is enforced. However, the code only enforces a 1-week pause.

```
///@dev Enforce a minimum of 2 weeks from deployment until distributing the
first loyalty reward
 if (_firstDistributionTime < block.timestamp + DISTRIBUTION_INTERVAL)
revert InvalidConstructor();
```

```
uint256 private constant DISTRIBUTION_INTERVAL = 604800; // 7 days between
loyalty reward distributions
```

Either fix the comment or the code to reflect the same initial distribution interval.

Remediation

Fixed comment.

[I-02] First trade runs for 2x TRADE_DURATION

On deployment, *activeCohortID* is set to 2, and *_firstSettlementTime* is defined as at least 2x *TRADE_DURATION* after creation of the contract.

This means that the first time *settleCohort* can be called will be 2x *TRADE_DURATION* from contract creation. This is in contrast to other cohorts, which only run for *TRADE_DURATION* amount of time.

```
if (_firstSettlementTime < block.timestamp + _tradeDuration * 2) revert
InvalidConstructor();
 nextSettlement = _firstSettlementTime;

 activeCohortID = 2;
```

Remediation

Fixed by removing the 2x factor.

[I-03] No pause mechanism on market

The *TopCutMarket* contract does not implement a pause mechanism. Since the protocol depends on a live and working oracle, it is recommended to implement a pause mechanism in the system. This way, in case the oracle breaks or the arbitrum sequencer goes down/suffers brief blackouts, future trades can be halted.

Remediation

Acknowledged.

[I-04] Oracle usage risks

Using chainlink oracles to settle trades has a few quirks, which can cause some unexpected results.

All chainlink oracles have a deviation threshold and heartbeat built into them Ref: https://data.chain.link/feeds/ethereum/mainnet/eth-usd.

Prices are only updated in the oracle if the price moves by more than the deviation threshold. The heartbeat parameter is the maximum amount of time that is allowed to pass before updating the price, regardless of the deviation threshold.

Due to the deviation threshold, prices can be inaccurate by up to 0.5% at the time of settlement for ETH/USDC feeds. Thus, traders are not actually betting on the actual ETH/USDC price but the Chainlink price, which is pushed on chain at that moment (which can be off by 0.5%).

In periods of low volatility, prices won't be continuously pushed to the oracle (if they remain within 0.5%). During the times, prices can be expected to be refreshed every 1 hour (heartbeat) for the WETH/USDC feed. In such scenarios, the protocol will be unusable.

This is because the new trades won't be accepted if the settlement timestamp has been crossed.

```
if (block.timestamp > settlementTime) revert WaitingToSettle();
```

But settlement cannot commence if the price hasn't been updated after the settlement timestamp.

```
if (updatedAt < settlementTime) revert StaleOraclePrice();
```

And due to the reason above, the protocol can wait up to 1 hour (heartbeat) for a price update to start settlement.

Thus, if the trade time is set to 24 hours, the protocol can be unusable for 1 hour every day due to the oracle heartbeat as a worst-case scenario.

The issue stated in point 1 is difficult to combat without a high degree of centralization. More accurate prices can be sourced from a combination of CEXs and DEXs and pushed on chain for settlement, which will get rid of the 0.5% error. However, since most defi protocols work with Chainlink and accept the 0.5% error, it can be accepted as an operational risk.

For 2, if the downtime is unacceptable, then a mechanism needs to be added where trades get added to a buffer while waiting for settlement. Once settled, the trades get added to the current available cohort.

Remediation

Acknowledged. Losing some time at the beginning of a prediction market due to pending settlements is acceptable.

[I-05] Some rewards are given on a first come first serve (FCFS) basis

If multiple users predict the same price value, winners will be chosen based on their trade order times.

For example, say the oracle price is 1000, and there are to be 5 winners. The predictions closest to it are [999,998,997,996,995,995,995,...]. While there are 3 users who predicted 995, only the first trader will be picked as the winner and rewarded with 10x. The others will not win even though they made the same choice.

While this is clearly a design decision, it should be communicated to the users clearly that in certain scenarios, the system reduces to a FCFS situation via the protocol docs.

Remediation

Acknowledged. Will be added to the docs and FAQ.

[I-06] Hardcoded treasury address

The *TopCutNFT* contract has a hardcoded treasury address. If the treasury is ever to be changed, it would require redeployment of the whole contract.

Consider making that parameter configurable by the admin.

```
address private constant TREASURY =
0xa0BFD02a7a47CBCA7230E03fbf04A196C3E771E3;
```

Remediation

Acknowledged.

[I-07] Unfinished IPFS URI

The *TopCutNFT* contract contains an incomplete IPFS URI.

```
string public metadataURI = "420g02n230f203f"; //////////
---------------------->>> UPDATE IPFS METADATA
```

Remediation

Acknowledged. It will be fixed before launch.

[I-08] tradesCohort not emptied out

After *settleCohort* is called, the data for that cohort is stale and should not be used for anything. This is reflected by setting the *cohortSize* of that cohort to 0, which is used in most places as the max iterator.

However, since *tradesCohort* is a mapping, it still retains the information from the last round. So if round 1 had 50 trades and then round 3 has 30 trades, the *tradesCohort* mapping will have elements 1-30 of the current trade and elements 31-50 of the previous already resolved trade.

There is no security issue in the current state; however, future changes or off-chain applications can still mistakenly use stale data if not careful.

Remediation

Acknowledged.

[I-09] Conflicting vault rewards

The vault gives out its collected eth in 3 ways:

1. 1% of the balance is given out as loyalty rewards.

2. A percentage is given out as affiliate rewards depending on the affiliate points of the caller
3. Users can claim a part of the ETH reserves by swapping in PSM tokens.

The issue is that the payout of all three methods is based on the current ETH balance of the contract. Because of this, claiming in any way lowers the rewards assigned for the other two methods. Thus, even unwilling transactions present in the same block can lead to unexpected results depending on the order of their execution.

Furthermore, the loyalty rewards distributed via *_distributeLoyaltyReward* do not even have any slippage parameter. So, if any user claims affiliate or PSM rewards, the loyalty rewards can drop drastically.

Claiming rewards in any way reduces rewards for others, so there is a race condition, and the rewards are dealt out on a first-come-first-serve basis.

Consider segregating rewards. That is, consider assigning a part of the rewards into a *loyaltyReward* bucket and a separate part into an *affiliateReward* bucket. This way, each party can claim its rewards without interference.

For PSM swaps, the affiliates and users might be credited with the accumulated PSM tokens so that they don't lose out on all their rewards.

Remediation

Acknowledged. Affiliates and PSM holders are expected to have common goals and a shared vision on the platform. Both also have access to slippage parameters.