SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	Predict.fun
Prepared by:	Sherlock
Lead Security Expert:	bughuntoor
Dates Audited:	October 1 - October 7, 2024
Prepared on:	October 30, 2024

Introduction

predict.fun is a yield-bearing prediction market built on Blast. This contest is for a peer to peer lending protocol that we are currently building.

Scope

Repository: PredictDotFun/predict-dot-loan

Branch: main

Audited Commit: a0e47f025761691fbbe174745faf61b966d77880

Final Commit: d5b82cfd5798540e420e5226842b322ec46d41ae

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
5	0

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

PUSH0
kuprum
valuevalk
bughuntoor
kennedy1030
iamnmt
000000
t.aksoy
infect3d

0xAadi
0xShoonya
KiroBrejka
ZanyBonzy
h2134
Ironsidesec
dany.armstrong90
tobi0x18
0xNirix

silver_eth
0xnbvc
0rpse
Pheonix
Sickurity
SyncCode2017
GGONE

Issue M-1: A borrower can not repay to a USDC black-listed lender

Source: https://github.com/sherlock-audit/2024-09-predict-fun-judging/issues/85

The protocol has acknowledged this issue.

Found by

000000, 0rpse, 0xnbvc, GGONE, PUSH0, Pheonix, Sickurity, SyncCode2017, bughuntoor, iamnmt, kennedy1030, t.aksoy, valuevalk

Summary

A borrower transfers `LOAN_TOKEN` directly to a lender when repaying their loan will cause the loan can not to be repaid when the lender is blacklisted by the `LOAN_TOKEN`.

Root Cause

A borrower repays their loan by transferring `LOAN_TOKEN` directly to a lender

https://github.com/sherlock-audit/2024-09-predict-fun/blob/41e70f9eed3f00dd29aba4038544150f5b35dccb/predict-dot-loan/contracts/PredictDotLoan.sol#L470

```
    function repay(uint256 loanId) external nonReentrant {
        Loan storage loan = loans[loanId];

        _assertAuthorizedCaller(loan.borrower);

        LoanStatus status = loan.status;
        if (status != LoanStatus.Active) {
            if (status != LoanStatus.Called) {
                revert InvalidLoanStatus();
            }
        }

        uint256 debt = _calculateDebt(loan.loanAmount,
↪   loan.interestRatePerSecond, _calculateLoanTimeElapsed(loan));

        loan.status = LoanStatus.Repaid;

>>      LOAN_TOKEN.safeTransferFrom(msg.sender, loan.lender, debt);
        CTF.safeTransferFrom(address(this), msg.sender, loan.positionId,
↪   loan.collateralAmount, "");
```

```
        emit LoanRepaid(loanId, debt);
    }
```

When `LOAN_TOKEN` is `USDC`, and the lender is blacklisted, the borrower can not transfer `USDC` to repay the lender.

Internal pre-conditions

1. `LOAN_TOKEN` is `USDC`
2. The borrower has borrowed from the lender before the lender is blacklisted by `USDC`.

External pre-conditions

The lender is blacklisted by `USDC`

Attack Path

1. The borrower borrows from the lender
2. The lender is blacklisted by `USDC`
3. The borrower can not repay the lender

Impact

1. The borrower can not repay their loan
2. The borrower can not get their collateral tokens (ERC1155) back
3. When the loan is matured, the lender can call the loan, and then seize all the collateral tokens (Note that, since the new lender also has to transfer `USDC` to the old lender, the auction functionality will not work, and the lender will guarantee to seize all the collateral tokens)

PoC

No response

Mitigation

Implement a pushing method for repaying a loan:

1. The borrower repays a loan by transferring the `LOAN_TOKEN` to the `PredictDotLoan` contract, and the loan amounts are credited to the lender.
2. The lender claims the loan amounts back from the `PredictDotLoan` contract.

Issue M-2: Collateral can already be seized even when negRiskMarket is not fully resolved

Source: https://github.com/sherlock-audit/2024-09-predict-fun-judging/issues/113

Found by

PUSH0

Summary

NegRiskMarket has a two step verification process, in order to ensure reported outcomes are correct. First the UMA oracle has the possibility to flag the answer and after this there is a period of time in which the negRiskAdapterOperator can flag the result.

View following code for negRiskOperator: NegRiskOperator.sol View following code for UMA report: UmaCtfAdapter.sol

The negRiskOperator can still change the answer in case he deems it to be incorrect, even after the UMA oracle has reported a valid outcome.

This leads to following problem: Currently the loan can be seized even if the negRiskAdapterOperator has flagged the result / the result is not yet determined.

In case the answer changes, it will lead to loss of collateral for the borrower.

Root Cause

Currently the _isQuestionPriceAvailable function checks if the UMA oracle OR the Market is determined. In case the UMA oracle returns a result, but this result is flagged and the market is not determined yet, the function will return true regardless.

```
function _isQuestionPriceAvailable(
        QuestionType questionType,
        bytes32 questionId
    ) private view returns (bool isAvailable) {
        if (questionType == QuestionType.Binary) {
            (isAvailable, ) =
↪   _isBinaryOutcomeQuestionPriceAvailable(UMA_CTF_ADAPTER, questionId);
        } else {
            (isAvailable, ) =
↪   _isBinaryOutcomeQuestionPriceAvailable(NEG_RISK_UMA_CTF_ADAPTER, questionId);
            isAvailable = isAvailable || _isNegRiskMarketDetermined(questionId);
```

```
        }
    }
```

https://github.com/sherlock-audit/2024-09-predict-fun/blob/main/predict-dot-loan/contracts/PredictDotLoan.sol#L1496C5-L1507C1

Internal pre-conditions

1. Lender creates loan on difficult market
2. The loan time ends and it becomes sizable

External pre-conditions

1. UMA Oracle returns an Answer
2. The UMA Oracles answers gets flagged / changed by the negRiskAdapterOperator.

Attack Path

1. Create loan on difficult market that ends shortly after the market resolves
2. Take unrightfully collateral in case outcome changes

Impact

Lender can take borrowers collateral even if the answer has not fully resolved. Breaking invariant and leading to loss of funds.

Mitigation

It should be sufficient to check only _isNegRiskMarketDetermined(questionId). From our research it can only return true in case the UMA oracle and the negRiskAdapter are correctly resolved.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/PredictDotFun/predict-dot-loan/pull/45

Issue M-3: Refinancing and auction take less fee than expected.

Source: https://github.com/sherlock-audit/2024-09-predict-fun-judging/issues/118

Found by

000000, 0xNirix, bughuntoor, dany.armstrong90, iamnmt, kennedy1030, silver_eth, t.aksoy, tobi0x18

Summary

When creating a new loan within `_acceptOffer`, the `protocolFee` is applied to the whole amount taken from the lender - the fulfilled amount.

```
function _transferLoanAmountAndProtocolFee(
    address from,
    address to,
    uint256 loanAmount
) private returns (uint256 protocolFee) {
    protocolFee = (loanAmount * protocolFeeBasisPoints) / 10_000;
    LOAN_TOKEN.safeTransferFrom(from, to, loanAmount - protocolFee);
    if (protocolFee > 0) {
        LOAN_TOKEN.safeTransferFrom(from, protocolFeeRecipient, protocolFee);
    }
}
```

So if the user fulfills 1000 USDC and the protocol fee is 2%, the fee that will be taken will be 20 USDC and the user will receive 980 USDC.

However, this is not the case within `refinance` and `auction`.

```
uint256 _nextLoanId = nextLoanId;
uint256 debt = _calculateDebt(loan.loanAmount, loan.interestRatePerSecond,
↪  callTime - loan.startTime);
uint256 protocolFee = (debt * protocolFeeBasisPoints) / 10_000;
```

There. the protocol fee is applied on the `debt`. So if a position with a debt of 980 USDC gets auctioned, the fee that will be paid will be 19,6 USDC. In this case, the protocol will earn 2% less fees than expected.

Whenever there's a protocol fee, `refinance` and `auction` will earn less fees proportional to the set `protocolFee`. Meaning that if the fee is 1%, these functions would earn 1% less. And if the fee is set to 2%, the loss will be 2%.

As the protocol can easily lose up to 2% of its fees, this according to Sherlock rules should be classified as High severity

> Definite loss of funds without (extensive) limitations of external conditions. The loss of the affected party must exceed 1%.

Root Cause

Wrong math formula used

Affected Code

https://github.com/sherlock-audit/2024-09-predict-fun/blob/main/predict-dot-loan/contracts/PredictDotLoan.sol#L585

Impact

Protocol will make significantly less fees than expected.

PoC

No response

Mitigation

Use the following formula instead

```
uint256 protocolFee = (debt * protocolFeeBasisPoints) / (10_000 -
↪  protocolFeeBasisPoints);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/PredictDotFun/predict-dot-loan/pull/50

Issue M-4: Using wrong format of `questionId` for `NegRiskCtfAdapter` leads to loan operations on resolved multi-outcome markets

Source: https://github.com/sherlock-audit/2024-09-predict-fun-judging/issues/119

Found by

kuprum

Summary

`PredictDotLoan` contract aims at preventing operations with loans as soon as underlying binary questions or multi-outcome markets become resolved. Unfortunately the determination of whether the multi-outcome markets are resolved is implemented incorrectly.

The problem is that though the format of `questionId`s employed in `UmaCtfAdapter` and `NegRiskAdapter` are different, they are treated as the same in `PredictDotLoan`; as a result of this misinterpretation, the request _isNegRiskMarketDetermined(bytes32 questionId) will always return `false`. This will lead to treating multi-outcome markets as unresolved, and thus to a guaranteed loss of funds: e.g. giving a loan to the borrow proposal for a position which is guaranteed to resolve to 0.

Root Cause

As outlined in the documentation for Polymarket Multi-Outcome Markets:

> The NegRiskOperator and NegRiskAdapter are designed to be used with the UmaCtfAdapter, or any oracle with the same interface. A dedicated UmaCtfAdapter will need to be deployed with the UmaCtfAdapter's `ctf` set to the address of the NegRiskAdapter, and the NegRiskOperator's `oracle` set to the address of the UmaCtfAdapter.
>
> In order to prepare a question for a market using the NegRiskOperator, the question must be initialized on the UmaCtfAdapter first. Then, the question may be prepared on the NegRiskOperator where the `_requestId` parameter is the `questionID` returned by the UmaCtfAdapter.

As can be seen, `questionId` as employed in `UmaCtfAdapter` becomes `_requestId` in `NegRiskOperator`, which generates its own `questionId`, in another format. Concretely:

- `UmaCtfAdapter`'s `questionId` is generated in UmaCtfAdapter::initialize as follows:

```
bytes memory data = AncillaryDataLib._appendAncillaryData(msg.sender,
↪  ancillaryData);
if (ancillaryData.length == 0 || data.length > MAX_ANCILLARY_DATA) revert
↪  InvalidAncillaryData();

questionID = keccak256(data);
```

Thus, this `questionId` is obtained by `keccak256` of initialization data.

- `NegRiskAdapter`'s `questionId` is generated via NegRiskOperator::prepareQuestion:

```
function prepareQuestion(bytes32 _marketId, bytes calldata _data, bytes32
↪  _requestId)
```

which then routes to MarketDataManager::_prepareQuestion:

```
function _prepareQuestion(bytes32 _marketId) internal returns (bytes32
↪  questionId, uint256 index) {
    MarketData md = marketData[_marketId];
    address oracle = marketData[_marketId].oracle();

    if (oracle == address(0)) revert MarketNotPrepared();
    if (oracle != msg.sender) revert OnlyOracle();

    index = md.questionCount();
    questionId = NegRiskIdLib.getQuestionId(_marketId, uint8(index));
    marketData[_marketId] = md.incrementQuestionCount();
}
```

As can be seen, the latter `questionId` is obtained by merging `marketId` (248 bits) and `index` (8 bits).

Despite this discrepancy in formats, the `questionId` from `UmaCtfAdapter` is employed in `PredictDotLoan` for requesting the state of the market from `NegRiskAdapter` in _assertQuestionPriceUnavailable:

```
function _assertQuestionPriceUnavailable(QuestionType questionType, bytes32
↪  questionId) private view {
    if (questionType == QuestionType.Binary) {
        _assertBinaryOutcomeQuestionPriceUnavailable(UMA_CTF_ADAPTER,
↪  questionId);
    } else {
        if (_isNegRiskMarketDetermined(questionId)) {
            revert MarketResolved();
```

```
        }
        _assertBinaryOutcomeQuestionPriceUnavailable(NEG_RISK_UMA_CTF_ADAPTER,
↪   questionId);
    }
}

function _isNegRiskMarketDetermined(bytes32 questionId) private view returns
↪   (bool isDetermined) {
@>>     isDetermined =
↪   NEG_RISK_ADAPTER.getDetermined(NegRiskIdLib.getMarketId(questionId));
}
```

`NegRiskAdapter`'s `getDetermined` is implemented as follows:

```
function getDetermined(bytes32 _marketId) external view returns (bool) {
    return marketData[_marketId].determined();
}

function determined(MarketData _data) internal pure returns (bool) {
    return MarketData.unwrap(_data)[1] == 0x00 ? false : true;
}
```

As `NegRiskIdLib.getMarketId` simply masks the last 8 bits away *from `questionId` in the wrong format*, and the above code simply reads the data from a mapping, combined it means that `getDetermined` will always return `false` as it will read data from an uninitialized mapping entry.

Impact

Guaranteed loss of funds: when a multi-outcome market gets resolved (e.g. we know that candidate A won elections), then all other positions (for candidates B, C, D) automatically become worthless. But if `PredictDotLoan` still treats the multi-outcome market as unresolved, this allows a multitude of exploits: e.g. grabbing an open loan proposal, and providing as collateral tokens for candidate B; or providing a loan for a still open borrow proposal for candidate A, and potentially seizing much more funds than the provided loan amount.

Mitigation

Apply the necessary missing step of indirection:

- Read the public questionIds mapping from `NegRiskOperator`, using `UmaCtfAdapter`'s `questionId` as `_requestId`:

  ```
  mapping(bytes32 _requestId => bytes32) public questionIds;
  ```

- Apply this value to request the market state in function _isNegRiskMarketDetermined.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/PredictDotFun/predict-dot-loan/pull/43

Issue M-5: hashProposal uses wrong typeshash when hashing the encoded Proposal struct data

Source: https://github.com/sherlock-audit/2024-09-predict-fun-judging/issues/266

Found by

0xAadi, 0xShoonya, Ironsidesec, KiroBrejka, ZanyBonzy, h2134, infect3d, valuevalk

Summary

`acceptLoanOfferAndFillOrder`, `_refinance`, `matchProposals` use `_assertValidSignature` which hashes proposal data and verifies the signature. But the hashed proposal type hash computation is wrong due to usage of `uint256 questionId` instead of `bytes32 questionId`

There are 2 impacts. So, even if one is acceptable/wrong, then the issue impact on another.

1. This will break the signature verification.
2. And breaking the strict EIP712's compatibility (mentioned in readme) where atomic types should be the same as the data format in the struct. Mentioned in `Definition of typed structured data` section.

Root Cause

Using `uint256 questionId` instead of `bytes32 questionId` inside the type hash of `hashProposal()`

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

Issue flow :

1. look at line 50 below, the `questionId` is in bytes 32. And when hashing a proposal data, the type hash of proposal struct format should also use bytes32 for question id. But here its using uint256. Check on line 819 below.

2. Due to this, the type hash will be different result. look at the chisel example below. The hashes are different, so the signature hash is using wrong digest to verify the signature. Should have used bytes32 itself.

This breaks the EIP712 , where atomic types like uint, bytes1 to bytes32, address should be directly used. And only strings, bytes data are dynamic types, should be keccack hashed and then derive the type hash.

https://github.com/sherlock-audit/2024-09-predict-fun/blob/41e70f9eed3f00dd29aba4038544150f5b35dccb/predict-dot-loan/contracts/interfaces/IPredictDotLoan.sol#L50

https://github.com/sherlock-audit/2024-09-predict-fun/blob/41e70f9eed3f00dd29aba4038544150f5b35dccb/predict-dot-loan/contracts/PredictDotLoan.sol#L817

```
IPredictDotLoan.sol

45:     struct Proposal {
    ---- SNIP ----
49:         QuestionType questionType;
50:   >   bytes32 questionId;
51:         bool outcome;
52:         uint256 interestRatePerSecond;
    ---- SNIP ----
59:         uint256 protocolFeeBasisPoints;
60:     }


PredictDotLoan.sol

814:     function hashProposal(Proposal calldata proposal) public view returns
 ↪  (bytes32 digest) {
815:         digest = _hashTypedDataV4(
816:             keccak256(
817:                 abi.encode(
818:                     keccak256(
819:      >>>                "Proposal(address from,uint256 loanAmount,uint256
 ↪  collateralAmount,uint8 questionType,uint256 questionId,bool outcome,uint256
 ↪  interestRatePerSecond,uint256 duration,uint256 validUntil,uint256
 ↪  salt,uint256 nonce,uint8 proposalType,uint256 protocolFeeBasisPoints)"
820:                     ),
    ---- SNIP ----
824:                     proposal.questionType,
825:                     proposal.questionId,
    ---- SNIP ----

834:                 )
```

```
835:             )
836:         );
837:     }
```

Impact

2 impacts

1. due to wrong type hash computation leading to wrong digest validation in the signature validator, the signatures might fail.
2. breaking the EIP712 mentioned in `readme` where it strictly complains. The atomic types should not be hashed or converted to other types.

PoC

No response

Mitigation

https://github.com/sherlock-audit/2024-09-predict-fun/blob/41e70f9eed3f00dd29aba4038544150f5b35dccb/predict-dot-loan/contracts/PredictDotLoan.sol#L817

```
    function hashProposal(Proposal calldata proposal) public view returns
↪   (bytes32 digest) {
        digest = _hashTypedDataV4(
            keccak256(
                abi.encode(
                   keccak256(
                       "Proposal(address from,uint256 loanAmount,uint256
↪   collateralAmount,uint8 questionType,
- uint256 questionId,
                       bool outcome,uint256 interestRatePerSecond,uint256
↪   duration,uint256 validUntil,uint256 salt,uint256 nonce,uint8
↪   proposalType,uint256 protocolFeeBasisPoints)"
                   ),
                   keccak256(
                        "Proposal(address from,uint256 loanAmount,uint256
↪   collateralAmount,uint8 questionType,
+ bytes32 questionId,
                   bool outcome,uint256 interestRatePerSecond,uint256
↪   duration,uint256 validUntil,uint256 salt,uint256 nonce,uint8
↪   proposalType,uint256 protocolFeeBasisPoints)"
                   ),
                   proposal.from,
    ---- SNIP ----
```

```
                )
            )
        );
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/PredictDotFun/predict-dot-loan/pull/37

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.