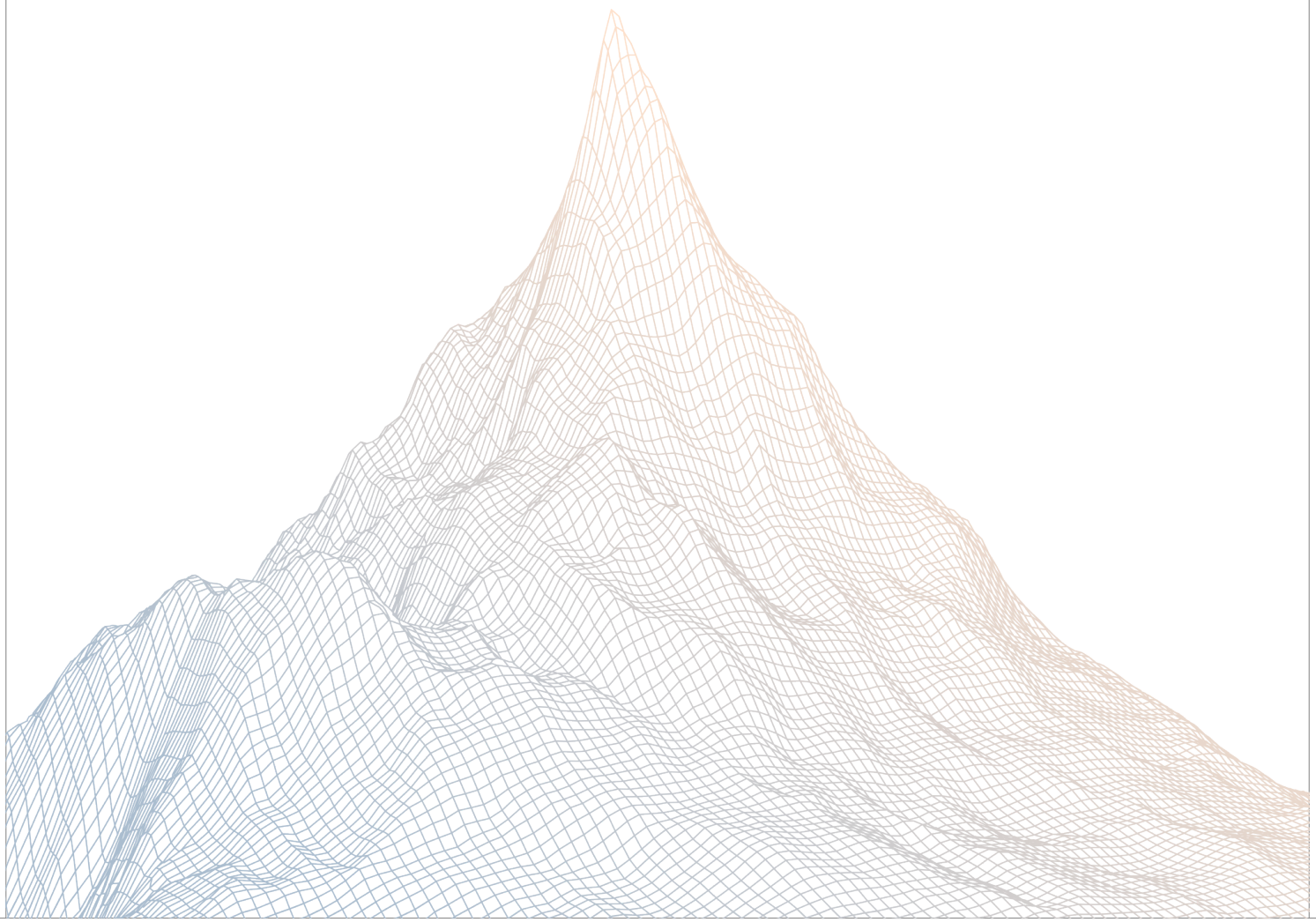
Zenith

Bonder

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: March 17th to March 19th, 2025

AUDITED BY: ether_sky
spicymeatball

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Bonder

Decentralized prediction market where users bet or provide liquidity for fees.

2.2 Scope

The engagement involved a review of the following targets:

Target	bonderShared
Repository	https://github.com/last0x/bonderShared
Commit Hash	`03108edd52239eb73f6ceb6722b4f1dc5528b99a`
Files	`src/modules/`

2.3 Audit Timeline

March 17, 2025	Audit start
March 19, 2025	Audit end
March 21, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	6
Medium Risk	5
Low Risk	2
Informational	2
Total Issues	**15**

3

Findings Summary

ID	Description	Status
H-1	The mintLpTokens function currently does not account for the creator and developer fees	Resolved
H-2	Restrict users from purchasing tokens until the liquidity provision process is completed	Acknowledged
H-3	USDC provided by liquidity providers can be distributed to betting winners	Acknowledged
H-4	There is no slippage check	Resolved
H-5	Possible underflow in sell functions	Resolved
H-6	Users can steal USDC from the pair by transferring LP tokens to the another address	Resolved
M-1	Creators can close the market and publish results anytime	Resolved
M-2	The already created markets can be broken when the USDC address is modified in the factory	Resolved
M-3	The Yes and No tokens must be added proportionally during the execution of the mintLpTokens function	Resolved
M-4	Any user has the ability to burn their tokens	Resolved
M-5	LP fees acquisition issue	Resolved
L-1	Payment tokens with non 6 decimals are not supported	Resolved
L-2	Small amounts of tokens can become stuck in the market	Acknowledged
I-1	Use SafeERC20 library in the market	Resolved
I-2	The text in the betTokenBurn function is incorrect	Resolved

4

Findings

4.1 High Risk

A total of 6 high risk findings were identified.

[H-1] The mintLpTokens function currently does not account for the creator and developer fees

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- BonderV1Pair.sol

Description:

Users can provide `liquidity` to the `market` and receive `LP tokens` based on the amount of `USDC` they provide. However, the `mintLpTokens` function does not account for the `creator` and `developer fees` when minting the `Yes` and `No` tokens.

- BonderV1Pair.sol#L387-L394

```
function mintLpTokens(
    uint256 _usdcAmount,
    address _to
) external lock returns (uint256 liquidity) {
    require(msg.sender == factory, "Only factory can call this function");
    // sufficient check

    betPool += _usdcAmount;

356:uint256 tokenSplitAmount = (_usdcAmount * 10 ** 18) / 10 ** 6 / 2;

    uint256 yesAmount = tokenSplitAmount;
    uint256 noAmount = tokenSplitAmount;

361:uint256 yesTokensWithPair = BonderV1BetToken(address(yesToken))
        .balanceOf(address(this));
363:uint256 noTokensWithPair = BonderV1BetToken(address(noToken)).balanceOf(
        address(this)
```

```
    );

    // maxYesTokensWithPair = yesTokensWithPair;
    // maxNoTokensWithPair = noTokensWithPair;

    uint256 _totalSupply = totalSupply; // gas savings, must be defined here
    since totalSupply can update in _mintFee
    if (_totalSupply == 0) {
372:    liquidity = Math.sqrt(yesAmount * noAmount) - MINIMUM_LIQUIDITY;
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first
    MINIMUM_LIQUIDITY tokens
    } else {
375:    liquidity = Math.min(
            (yesAmount * _totalSupply) / yesTokensWithPair,
            (noAmount * _totalSupply) / noTokensWithPair
        );
    }
    require(liquidity > 0, "Insufficient liquidity mined");

    // reserves need to be manually updated as it can increase without tokens
    being minted (cannot use balanceOf)
    yesReserve += yesAmount;
    noReserve += noAmount;

    // mint bet tokens to pair contract, telling us these tokens have claims
    to the bet pool
387:BonderV1BetToken(address(yesToken)).betTokenMint(
        address(this),
        yesAmount
    );
391:BonderV1BetToken(address(noToken)).betTokenMint(
        address(this),
        noAmount
    );

    // mint lp tokens to caller
    _mint(_to, liquidity);
}
```

For example:

- Suppose a user provides `1000 USDC`. The `tokenSplitAmount` would be `500` (`line 356`), and the user would receive approximately `500 LP tokens` (ignoring `MINIMUM_LIQUIDITY`) at `line 372`. Then, `500 Yes` and `No` tokens would be minted (`lines 387` and `391`).
- However, if the total `creator` and `developer fees` amount to `10%`, only 450 `Yes` and `No` tokens would actually be minted to the `market`.

Now, if a second user provides another `1000 USDC`:

- The `tokenSplitAmount` remains the same at `500`.
- The `yesTokensWithPair` and `noTokensWithPair` values would be `450` (`lines 361` and `363`).
- This user would receive `liquidity` calculated as `500×500/450=555 LP tokens` at `line 375`.

This creates an unfair situation where both users provide the same amount of `USDC` but receive different amounts of `LP tokens`, which is inconsistent with normal protocols. The discrepancy arises because the `creator` and `developer fees` are not considered in the calculations.

Recommendations:

The simplest solution is to track the total amount of deposited tokens and use that value instead of relying on the current `balance`.

Bonder: Fixed in @870ee31924...

Zenith: Verified. The pair doesn't use betting token balances for LP amount calculation; it relies on the `lpPool` and its own total supply instead.

[H-2] Restrict users from purchasing tokens until the liquidity provision process is completed

SEVERITY: High	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: High

Target

- BonderV1AMMFactory.sol

Description:

Users can provide `liquidity` until the `liquidityEndTime`.

- BonderV1AMMFactory.sol#L320

```
function addLiquidity(uint256 _betId, uint256 _usdcAmount)
    external whenNotPaused returns (uint256 liquidity) {
    Bet storage bet = idToBet[_betId];

    require(block.timestamp < bet.liquidityEndTime, "Can only add liquidity
    before liquidity end time");
}
```

And purchase `Yes` or `No` tokens at any point until the `marketEndTime`.

- BonderV1AMMFactory.sol#L370

```
function _buy(uint256 _betId, uint256 _usdcAmount, uint8 _yesNo) private {
   Bet storage bet = idToBet[_betId];

   require(block.timestamp < bet.marketEndTime, "Can only bet before
    marketEndTime");
}
```

However, this can leads to several issues:

- `Buyers` may receive `0` tokens if `liquidity` has not yet been added.
- BonderV1Pair.sol#L186

```
function mintBetTokens(
    uint256 _usdcAmount,
    uint8 _yesNo,
    address _to
) external lock returns (uint256 amountReceived) {
    if (_yesNo == 1) {
        // For buying Yes tokens:
        // 1. Calculate new Yes tokens needed to maintain ratio after adding
    USDC value to No reserve
        yesBets += buyAmountNetFeeIn18Decimals;
        uint256 newNoReserve = noReserve + buyAmountNetFeeIn18Decimals;
        uint256 newYesReserve = (yesReserve * noReserve) / newNoReserve;
        // 2. The amount received is the difference
@->         amountReceived = yesReserve - newYesReserve;
    } else {
        // For buying No tokens:
        // 1. Calculate new No tokens needed to maintain ratio after adding
    USDC value to Yes reserve
        noBets += buyAmountNetFeeIn18Decimals;
        uint256 newYesReserve = yesReserve + buyAmountNetFeeIn18Decimals;
        uint256 newNoReserve = (noReserve * yesReserve) / newYesReserve;
        // 2. The amount received is the difference
@->         amountReceived = noReserve - newNoReserve;
    }
}
```

- Token prices adjust dynamically based on demand—higher demand causes prices to increase. Regardless of this, users providing liquidity can potentially acquire more tokens than those purchasing them, which creates an unfair advantage.
- BonderV1Pair.sol#L375-L378

```
function mintLpTokens(
    uint256 _usdcAmount,
    address _to
) external lock returns (uint256 liquidity) {
    uint256 tokenSplitAmount = (_usdcAmount * 10 ** 18) / 10 ** 6 / 2;

    uint256 yesAmount = tokenSplitAmount;
    uint256 noAmount = tokenSplitAmount;

    uint256 yesTokensWithPair = BonderV1BetToken(address(yesToken))
        .balanceOf(address(this));
    uint256 noTokensWithPair = BonderV1BetToken(address(noToken)).balanceOf(
        address(this)
    );
```

```
    uint256 _totalSupply = totalSupply; // gas savings, must be defined here
    since totalSupply can update in _mintFee
    if (_totalSupply == 0) {
        liquidity = Math.sqrt(yesAmount * noAmount) - MINIMUM_LIQUIDITY;
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first
    MINIMUM_LIQUIDITY tokens
    } else {
        liquidity = Math.min(
            (yesAmount * _totalSupply) / yesTokensWithPair,
            (noAmount * _totalSupply) / noTokensWithPair
        );
    }
    require(liquidity > 0, "Insufficient liquidity mined");

    _mint(_to, liquidity);
}
```

Recommendations:

```
function _buy(uint256 _betId, uint256 _usdcAmount, uint8 _yesNo) private {
    Bet storage bet = idToBet[_betId];


    require(bet.maxBetAmount == 0 || _usdcAmount <= bet.maxBetAmount, "Bet
    amount exceeds max bet amount");

    require(!bet.isClosed, "Can only bet when market is open");
    require(block.timestamp < bet.marketEndTime, "Can only bet before
    marketEndTime");

+^^Irequire(block.timestamp >= bet.liquidityEndTime, "Can only bet after
    liquidityEndTime");
}
```

Bonder: Acknowledged.

Zenith: Traders can use the newly added minimum amounts received check to protect themselves from this issue.

[H-3] USDC provided by liquidity providers can be distributed to betting winners

SEVERITY: High	IMPACT: High
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- BonderV1Pair.sol#L354

Description:

When liquidity providers deposit USDC into the pair contract to mint betting tokens, the funds are added to the betting pool:

```
function mintLpTokens(
    uint256 _usdcAmount,
    address _to
) external lock returns (uint256 liquidity) {
    require(msg.sender == factory, "Only factory can call this function");
    // sufficient check

    betPool += _usdcAmount;
```

Since LPs receive an equal amount of Yes and No tokens when adding liquidity, they may not be able to fully recover their USDC after the betting period ends. Some of their funds could be redistributed to the winners:

```
        if (result == 1) {
            // if result is set, then we can claim the bet tokens
            tokenBalance
    = BonderV1BetToken(address(yesToken)).balanceOf(_to);
            require(tokenBalance > 0, "No 'Yes' claims");

            // winningClaim in 6 dp
>>          winningClaim = (tokenBalance * betPool) / totalYesMinted;
            BonderV1BetToken(address(yesToken)).betTokenBurn(_to,
    tokenBalance);
```

Recommendations:

It might be better to move the USDC received from liquidity providers into a separate pool, so LP holders can fully retrieve their funds based on the amount of LP tokens they hold.

Bonder: Acknowledged as a won't fix.

Zenith: Liquidity providers should assess their own risks before entering the market.

[H-4] There is no slippage check

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- BonderV1Pair.sol

Description:

Users can purchase `Yes` or `No` tokens using `USDC`. The amount they can buy depends on the current `reserves` of both tokens, following a mechanism similar to `swaps` in standard `DEXs` like `Uniswap`.

- BonderV1Pair.sol#L161

```
function mintBetTokens(
    uint256 _usdcAmount,
    uint8 _yesNo,
    address _to
) external lock returns (uint256 amountReceived) {
    if (_yesNo == 1) {
        yesBets += buyAmountNetFeeIn18Decimals;
        uint256 newNoReserve = noReserve + buyAmountNetFeeIn18Decimals;
        uint256 newYesReserve = (yesReserve * noReserve) / newNoReserve;
@->         amountReceived = yesReserve - newYesReserve;
    } else {
    }
}
```

It is essential that all `swap` functions include `slippage protection`. Without it, users may receive significantly fewer tokens than expected, exposing them to well-known exploits.

Furthermore, the `mintLpTokens` function also lacks a `slippage check`.

- BonderV1Pair.sol#L351

```
function mintLpTokens(
    uint256 _usdcAmount,
    address _to
```

```
) external lock returns (uint256 liquidity) {
    uint256 _totalSupply = totalSupply; // gas savings, must be defined here
    since totalSupply can update in _mintFee
    if (_totalSupply == 0) {
        liquidity = Math.sqrt(yesAmount * noAmount) - MINIMUM_LIQUIDITY;
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first
    MINIMUM_LIQUIDITY tokens
    } else {
        liquidity = Math.min(
            (yesAmount * _totalSupply) / yesTokensWithPair,
            (noAmount * _totalSupply) / noTokensWithPair
        );
    }
    require(liquidity > 0, "Insufficient liquidity mined");

    _mint(_to, liquidity);
}
```

Recommendations:

Implement `slippage protection` mechanisms in the `mintBetTokens` and `mintLpTokens` functions.

Bonder: Resolved with @0323866a12ed...

Zenith: Verified. Added slippage checks for buy/sell functions.

[H-5] Possible underflow in sell functions

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- BonderV1Pair.sol#L299
- BonderV1Pair.sol#L286

Severity:

- Impact: High
- Likelihood: Medium

Description:

Sell functions in the bonder pair can be blocked due to an underflow. The issue is that the contract subtracts USDC amounts from the `yesBets` and `noBets` variables, which may be smaller than the required amount due to the current state of the AMM. Steps to reproduce:

- Disable all fees for simplicity, betId = 1
- addLiquidity(1, 10_000e6)
- userA -> buyYes(1, 1000e6)
- userB -> buyNo(1, 1000e6)
- userA -> sellYes(1, userA balance)
- userB -> sellNo(1, userB balance)

When userB attempts to sell all his NO tokens (approx 1161.2) `noBets` variable will underflow ,state at the moment of revert: `noBets` = 1000e18 `receivedUsdcAmount` = 1307e18

Recommendations:

Consider removing the tracking of `yesBets` and `noBets`, as they are only used for informational purposes.

Bonder: Fixed in @d919064a6e4...

Zenith: Verified.

[H-6] Users can steal USDC from the pair by transferring LP tokens to the another address

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- BonderV1Pair.sol#L493-L494
- BonderV1Pair.sol#L433

Severity:

- Impact: High
- Likelihood: High

Description:

The collected fees are calculated as the difference between LP shares and the initial deposit:

```
uint256 lpDeposits = lpDeposits[_to];
uint256 fees = lpShare - lpDeposits;
```

If a user transfers LP tokens to an address with lpDeposits = 0, they can exploit this logic to claim both their deposit as a reward (fees = lpShare - 0) and the tokens they would normally receive from a winning claim.

Recommendations:

It is possible to restrict the transfer of LP tokens by making them soulbound.

Bonder: Fixed in @00f52516de0788d...

Zenith: Verified. LP tokens can be transferred only to whitelisted addresses.

4.2 Medium Risk

A total of 5 medium risk findings were identified.

[M-1] Creators can close the market and publish results anytime

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- BonderV1AMMFactory.sol#L445-L465

Description:

When a market is created, specific timing constraints are set for trading and liquidity phases. These ensure an orderly process where liquidity providers first supply funds, and users place bets afterward. However, the market creator can close the market and publish results at any stage, including during active trading.

This poses a risk because results can be seen in the mempool, allowing resourceful bettors to frontrun the transaction and place winning bets.

Recommendations:

Restrict result publication until the market has reached it's end time (betting no longer allowed).

Bonder: Fixed in @875070fdf9d...

Zenith: Verified. Creators can close the market in the open market using the `emergencyClose` function, but need to publish the results separately by calling the `closeMarket` function. There are no checks to confirm if trading is active, so it must be used with caution.

[M-2] The already created markets can be broken when the USDC address is modified in the factory

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- BonderV1AMMFactory.sol

Description:

Based on the comments, `USDC tokens` in the `markets` can potentially be replaced with other tokens.

- BonderV1Pair.sol#L56

```
// potential to be other tokens other than usdc
address public usdcAddress;
```

There is also a function in the `factory` that allows the `USDC address` to be modified.

- BonderV1AMMFactory.sol#L114-L121

```
function setUSDC(address newUSDC_) external {
    require(
        msg.sender == controller,
        "Only controller can set usdc address"
    );
    usdcAddress = newUSDC_;
    usdc = IERC20(newUSDC_);
}
```

When a `market` is created, the current `USDC address` of the `factory` is assigned to it and remains fixed.

- BonderV1Pair.sol#L99

```
constructor(
    ...
```

```
    address _usdcAddress,
    ...
) BonderV1ERC20(
    string(abi.encodePacked("Bonder-Pair-", Strings.toString(_contractId),
    "-", Strings.toString(_betCount))),
    string(abi.encodePacked("Bonder-Pair-", Strings.toString(_contractId),
    "-", Strings.toString(_betCount))),
    18
) {
    factory = msg.sender;
    usdcAddress = _usdcAddress;
    usdc = IERC20(_usdcAddress);
    feeRate = _feeRate;
}
```

However, if the `USDC address` is updated in the `factory`, future operations like `buy` and `addLiquidity` will use the new `USDC address`. Meanwhile, the already created `markets` will continue operating with the old `USDC address`.

This inconsistency will lead to a breakdown in the functionality of the existing `markets`, as they are no longer aligned with the updated `USDC`.

Recommendations:

Include a `USDC address` in the `Bet` struct to represent the `market-specific USDC`. Ensure that all operations in the `factory` reference this `USDC address` for the respective `market`.

Bonder: Fixed in @74ba88ee2641...

Zenith: Verified. The USDC address can no longer be changed.

[M-3] The Yes and No tokens must be added proportionally during the execution of the mintLpTokens function

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- BonderV1Pair.sol

Description:

Consider a scenario where LPs provide USDC, and the current market balance for Yes and No tokens is 1000 each, with a total token supply of 1000. Now, assume the balance of Yes tokens increases to 1200 due to external actions, such as the creator or developer sending their Yes tokens directly to the market.

- BonderV1Pair.sol#L375-L378

```
function mintLpTokens(
    uint256 _usdcAmount,
    address _to
) external lock returns (uint256 liquidity) {
    require(msg.sender == factory, "Only factory can call this function");
    // sufficient check

    betPool += _usdcAmount;

356:    uint256 tokenSplitAmount = (_usdcAmount * 10 ** 18) / 10 ** 6 / 2;

    uint256 yesAmount = tokenSplitAmount;
    uint256 noAmount = tokenSplitAmount;

361:    uint256 yesTokensWithPair = BonderV1BetToken(address(yesToken))
        .balanceOf(address(this));
363:    uint256 noTokensWithPair
    = BonderV1BetToken(address(noToken)).balanceOf(
        address(this)
    );

    // maxYesTokensWithPair = yesTokensWithPair;
```

```
    // maxNoTokensWithPair = noTokensWithPair;

    uint256 _totalSupply = totalSupply; // gas savings, must be defined here
    since totalSupply can update in _mintFee
    if (_totalSupply == 0) {
        liquidity = Math.sqrt(yesAmount * noAmount) - MINIMUM_LIQUIDITY;
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first
    MINIMUM_LIQUIDITY tokens
    } else {
375:        liquidity = Math.min(
            (yesAmount * _totalSupply) / yesTokensWithPair,
            (noAmount * _totalSupply) / noTokensWithPair
        );
    }
    require(liquidity > 0, "Insufficient liquidity mined");

    // reserves need to be manually updated as it can increase without tokens
    being minted (cannot use balanceOf)
    yesReserve += yesAmount;
    noReserve += noAmount;

    // mint bet tokens to pair contract, telling us these tokens have claims
    to the bet pool
387:    BonderV1BetToken(address(yesToken)).betTokenMint(
        address(this),
        yesAmount
    );
391:    BonderV1BetToken(address(noToken)).betTokenMint(
        address(this),
        noAmount
    );

    // mint lp tokens to caller
    _mint(_to, liquidity);
}
```

A user then decides to add `2000 USDC` as `liquidity`:

- The `tokenSplitAmount` is calculated as `1000` (`line 356`).
- The `yesTokensWithPair` is `1200` (`line 361`), and the `noTokensWithPair` is `1000` (`line 363`).
- The `liquidity` would be calculated as `1000×1000/1200=833` at `line 375`.
- After the transaction, the `balance` of `Yes` tokens increases to `2200` (`line 387`), and the `balance` of `No` tokens increases to `2000` at `line 391`.

Next, if the user burns these `833 liquidity` tokens:

- They would receive `833×2200/1833=1000 Yes` tokens and `833×2000/1833=908 No` tokens.

This results in a loss of `No` tokens for the user. Ideally, for the `833 liquidity`, the `No` tokens added should have been `833×1000/1000=833`, to ensure proportional distribution. In this case, the user would receive `833×2200/1833=1000 Yes` tokens and `833×1833/1833=833 No` tokens, which accurately reflects the amount they initially provided.

Recommendations:

```
function mintLpTokens(
    uint256 _usdcAmount,
    address _to
) external lock returns (uint256 liquidity) {
    uint256 tokenSplitAmount = (_usdcAmount * 10 ** 18) / 10 ** 6 / 2;

    uint256 refundUSDC = 0;

    uint256 _totalSupply = totalSupply; // gas savings, must be defined here
    since totalSupply can update in _mintFee
    if (_totalSupply == 0) {
        liquidity = Math.sqrt(yesAmount * noAmount) - MINIMUM_LIQUIDITY;
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first
    MINIMUM_LIQUIDITY tokens
    } else {
        liquidity = Math.min(
            (yesAmount * _totalSupply) / yesTokensWithPair,
            (noAmount * _totalSupply) / noTokensWithPair
        );
        yesAmount = liquidity * yesTokensWithPair / _totalSupply;
        noAmount = liquidity * noTokensWithPair / _totalSupply;
        refundUSDC += (tokenSplitAmount - yesAmount);
        refundUSDC += (tokenSplitAmount - noAmount);
        refundUSDC /= 10 ** 12;
    }
    require(liquidity > 0, "Insufficient liquidity mined");

    // reserves need to be manually updated as it can increase without tokens
    being minted (cannot use balanceOf)
    yesReserve += yesAmount;
    noReserve += noAmount;

    // mint bet tokens to pair contract, telling us these tokens have claims
    to the bet pool
    BonderV1BetToken(address(yesToken)).betTokenMint(
        address(this),
```

```
        yesAmount
    );
    BonderV1BetToken(address(noToken)).betTokenMint(
        address(this),
        noAmount
    );

    // mint lp tokens to caller
    _mint(_to, liquidity);
    if (refundUSDC != 0) {
        require(usdc.transfer(_to, refundUSDC), "Usdc transfer failed");
    }
    }
```

Bonder: Fixed in @88d536b74ed...

Zenith: Verified. The LP allocation mechanism has changed; it now mints LP tokens based on USDC supply and collected fees.

[M-4] Any user has the ability to burn their tokens

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- BonderV1ERC20.sol

Description:

The `BonderV1ERC20` contract includes a specific `_burn` function, allowing tokens to be burned.

- BonderV1ERC20.sol#L62-L64

```
function _burn(address account, uint256 value) internal {
    _update(account, address(0), value);
}
```

And the `betTokenBurn` function permits only the `owner` to `burn` tokens on behalf of others.

- BonderV1BetTokenWithFee.sol#L60-L63

```
function betTokenBurn(address _account, uint256 _amount) public {
        require(msg.sender == owner, "Only owner can mint");
        _burn(_account, _amount);
}
```

However, there is no restriction in the `_transfer` function to ensure that the `from` and `to` addresses are not `address(0)`.

- BonderV1ERC20.sol#L66-L68

```
function _transfer(address from, address to, uint256 value) internal {
    _update(from, to, value);
}
```

As a result, any user can `burn` their tokens by transferring them to `address(0)`. This could lead to unintended consequences, such as users burning tokens and `locking` the

corresponding `USDC` in the market. The `feePool` and `betPool` in the `market` are distributed to `holders` of `Yes` and `No` tokens based on their `balances`.

Recommendations:

```
function _transfer(address from, address to, uint256 value) internal {
    require(from ≠ address(0), "");
    require(to ≠ address(0), "");
    _update(from, to, value);
    }
```

Bonder: Fixed in @11d50ae5320...

Zenith: Verified. A check has been added to prevent transfers to the zero address.

[M-5] LP fees acquisition issue

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- BonderV1Pair.sol#L401-L438

Description:

As described by the client, there is a known issue with how the Bonder pair distributes fees to LP holders. Currently, anyone can mint a large number of LP tokens and claim previously collected fees for themselves, which contradicts the Uniswap-like design of the contract. There needs to be a solution to distribute fees collected from bet token purchases to LP holders who hold tokens at the time of fee collection

Recommendations:

One way to address this issue is to separate LP holders from the betting pool and create a dedicated pool for their USDC and collected fees:

```
    function mintLpTokens(
        uint256 _usdcAmount,
        address _to
    ) external lock returns (uint256 liquidity) {
        require(msg.sender == factory, "Only factory can call this
    function"); // sufficient check


        uint256 tokenSplitAmount = (_usdcAmount * 10 ** 18) / 10 ** 6 / 2;

        uint256 yesAmount = tokenSplitAmount;
        uint256 noAmount = tokenSplitAmount;

        uint256 yesTokensWithPair = BonderV1BetToken(address(yesToken))
            .balanceOf(address(this));
        uint256 noTokensWithPair = BonderV1BetToken(address(noToken)).
            balanceOf(
```

```
            address(this)
        );

        // maxYesTokensWithPair = yesTokensWithPair;
        // maxNoTokensWithPair = noTokensWithPair;

        uint256 _totalSupply = totalSupply; // gas savings, must be defined
here since totalSupply can update in _mintFee
        if (_totalSupply == 0) {
            /// audit
            liquidity = _usdcAmount - MINIMUM_LIQUIDITY;

    liquidity = Math.sqrt(yesAmount * noAmount) - MINIMUM_LIQUIDITY;
            _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the
first MINIMUM_LIQUIDITY tokens
        } else {
            /// audit
            liquidity = _usdcAmount * _totalSupply / lpPool;
            liquidity = Math.min(
                (yesAmount * _totalSupply) / yesTokensWithPair,
                (noAmount * _totalSupply) / noTokensWithPair
            );
        }
        require(liquidity > 0, "Insufficient liquidity mined");

        // reserves need to be manually updated as it can increase without
tokens being minted (cannot use balanceOf)
        yesReserve += yesAmount;
        noReserve += noAmount;
      lpPool += _usdcAmount;

        // mint bet tokens to pair contract, telling us these tokens have
claims to the bet pool
        // audit no need for this
        BonderV1BetToken(address(yesToken)).betTokenMint(
            address(this),
            yesAmount
        );
        BonderV1BetToken(address(noToken)).betTokenMint(
            address(this),
            noAmount
        );
```

```
        // mint lp tokens to caller
        _mint(_to, liquidity);
    }
```

In the modified function, LP tokens are now minted solely based on the amount of USDC provided and the current amount of collected fees. The `betPool` is no longer updated with USDC; instead, the `feePool`, renamed to `lpPool`, is updated. Additionally, the minting of betting tokens to `address(this)` has been removed, and only virtual reserves are updated.

```
    function burnLpTokens(
        address _to
    ) external lock returns (uint256 yesAmount, uint256 noAmount) {
        require(msg.sender == factory, "Only factory can call this
    function"); // sufficient check

        // what has been transferred to the pair contract
        // liquidity, _totalSupply, yesAmount, noAmount are in 18 dp
        uint256 liquidity = balanceOf[address(this)];
        uint256 _totalSupply = totalSupply;

        uint256 yesTokensWithPair = BonderV1BetToken(address(yesToken))
            .balanceOf(address(this));
        uint256 noTokensWithPair = BonderV1BetToken(address(noToken)).
            balanceOf(
            address(this)
        );
        uint256 _totalMinted = totalMinted;

        // totalMinted = BonderV1BetToken(address(yesToken)).totalMinted();

        yesAmount = (liquidity * yesTokensWithPair) / _totalSupply;
        // using balances ensures pro-rata distribution
        noAmount = (liquidity * noTokensWithPair) / _totalSupply;
        // using balances ensures pro-rata distribution
        require(yesAmount > 0 && noAmount > 0, "Insufficient liquidity
            burned");

        // transfer from its own balance that was minted previously
        require(yesToken.transfer(_to, yesAmount), "Yes token transfer
            failed");
```

```
    require(noToken.transfer(_to, noAmount), "No token transfer failed");

    // transfer fees to lp holder
    // feePool is in 6 dp. liquidity and totalSupply are in 18 dp -
cancel out
    uint256 lpAmount = (lpPool * liquidity) / _totalMinted;

    // feePool -= feePoolAmount;

    require(usdc.transfer(_to, feePoolAmount), "Usdc transfer failed");

    _burn(address(this), liquidity); // burn liquidity amount from this
}
```

With this approach, fees are distributed only to LP holders who were holding LP at the time the fees were collected. New LP minters will receive only what they contributed, preventing them from unfairly claiming previously accrued fees.

Bonder: Fixed in @88d536b74ed...

Zenith: Verified. The LP allocation mechanism has changed; it now mints LP tokens based on USDC supply and collected fees. This ensures the accurate tracking and distribution of collected fees to LP token holders.

4.3 Low Risk

A total of 2 low risk findings were identified.

[L-1] Payment tokens with non 6 decimals are not supported

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- BonderV1Pair.sol#L172
- BonderV1Pair.sol#L356

Description:

The Bonder pair contract assumes the payment token has 6 decimals, like USDC:

```
    function mintBetTokens(
        uint256 _usdcAmount,
        uint8 _yesNo,
        address _to
    ) external lock returns (uint256 amountReceived) {
        --- SNIP ---
        // convert to 18 decimals
>>      uint256 buyAmountNetFeeIn18Decimals = (buyAmountNetFee * 10 ** 18)
    / 10 ** 6;
```

However, comments suggest other tokens might be used:

```
contract BonderV1Pair is BonderV1ERC20 {
    address public factory;
    BonderV1BetToken public yesToken;
    BonderV1BetToken public noToken;
    // potential to be other tokens other than usdc
>>  address public usdcAddress;
```

If a token with 18 decimals is used instead, scaling calculations may be incorrect.

Recommendations:

Use the `decimals()` value for proper scaling:

```
uint8 paymentTokenDecimals = IERC20(paymentToken).decimals();
uint256 buyAmountNetFeeIn18Decimals = (buyAmountNetFee * 10 ** 18) / 10 **
    paymentTokenDecimals;
```

Bonder: Fixed in @74ba88ee2641...

Zenith: Verified. The `decimals()` value is used instead of a hardcoded number.

[L-2] Small amounts of tokens can become stuck in the market

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- BonderV1Pair.sol

Description:

When the initial `LP` adds `liquidity`, the `MINIMUM_LIQUIDITY` (1000) is assigned to `address(0)`.

- BonderV1Pair.sol#L373

```
function mintLpTokens(
    uint256 _usdcAmount,
    address _to
) external lock returns (uint256 liquidity) {

    uint256 _totalSupply = totalSupply; // gas savings, must be defined here
    since totalSupply can update in _mintFee
    if (_totalSupply == 0) {
        liquidity = Math.sqrt(yesAmount * noAmount) - MINIMUM_LIQUIDITY;
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first
    MINIMUM_LIQUIDITY tokens
    } else {
        liquidity = Math.min(
            (yesAmount * _totalSupply) / yesTokensWithPair,
            (noAmount * _totalSupply) / noTokensWithPair
        );
    }
}
```

Consequently, the associated `USDC`, `Yes`, and `No` tokens tied to this `liquidity` become permanently stuck in the `market`. Additionally, minor discrepancies in calculations can result in dust amounts accumulating over time.

Recommendations:

The `factory` can claim the this `MINIMUM_LIQUIDITY liquidity` if all other `liquidities` have been burned.

Bonder: Acknowledged

4.4 Informational

A total of 2 informational findings were identified.

[I-1] Use SafeERC20 library in the market

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- BonderV1Pair.sol

Description:

Based on the comments, `USDC tokens` in the `markets` can potentially be replaced with other tokens.

- BonderV1Pair.sol#L56

```
// potential to be other tokens other than usdc
address public usdcAddress;
```

The `USDC transfer` function is designed to return a `boolean` value. However, it may not always true because the `USDC` can be changed to other token.

- BonderV1Pair.sol#L343

```
function claimWinnings(
    address _to
) external lock returns (uint256 winningClaim) {

    require(usdc.transfer(_to, winningClaim), "Usdc transfer failed");
    // if result is set, then we can claim the bet tokens
}
```

Recommendations:

Use `SafeERC20` library for safety.

Bonder: Fixed in @52b0473aa8de..., @e4b79f276e32...

Zenith: Verified. The SafeERC20 library is used for token transfers in the pair and factory contracts.

[I-2] The text in the betTokenBurn function is incorrect

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- BonderV1BetTokenWithFee.sol

Description:

The error message in the `require` statement within the `betTokenBurn` function is incorrect.

```
function betTokenBurn(address _account, uint256 _amount) public {
        require(msg.sender == owner, "Only owner can mint");
        _burn(_account, _amount);
}
```

Recommendations:

```
function betTokenBurn(address _account, uint256 _amount) public {
        require(msg.sender == owner, "Only owner can mint");
          require(msg.sender == owner, "Only owner can burn");
        _burn(_account, _amount);
        }
```

Bonder: Fixed in @ff4e213ccdbbd...

Zenith: Verified.