Introduction

A smart contract security audit review of thruster protocol was done by sparkware's auditor, 0xladboy233, with a focus on the security aspects of the application's implementation.

About Sparkware security

Sparkware security offers cutting edge and affordable smart contract auditing solution. The auditors get reputatoin and skill by consistently get top place in bug bounty and audit competition.

Our past audit prject including optimism and notional finance and graph protocol. Below is a list of comprehensive security review and research: https://github.com/JeffCX/Sparkware-audit-portfolio

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

About **Protocol Name**

Thruster is a dex on thruster, supporting both Uniswap V3 style concentrated liquidity and Uniswap V2 style liquidity curve.

Security Assessment Summary

review commit hash **-** **0568a06ed869931bc5ed478add10c873abfd9c78**

Scope

The following smart contracts were in scope of the audit:

- `thruster-cfmm/*`
- `thruster-clmm/*`
- thruster-treasure/*

Issue ID	Title
M1	PoolDeployer's gas yield cannot be properly claimed
M2	Winning users can lose their reward and prize depends on winning tickets and max count configuration
L1	Yield cut is not initialized in cfmm factory
L2	Entropy provider address in Thruster treasury should be changeable
L3	USDB / WETHB contract can be changed in mainnet
L4	Incorrect chain id check in NonfungibleTokenPositionDescriptor.sol implementation
R1	Yield claim mechanism can be more efficient
R2	Consider set a yield governor to claim the yield for protocol to reduce the contract size
R3	revealRandomNumber function should have internal modifier instead of public modifier
R4	User claim reward process can be more gas efficient

M1 – PoolDeployer's gas yield cannot be properly claimed

Description

When admin claim gas yield via PoolDeployer, the gas yield to claim is from contract address(0), but the poolDeployer is not authorized to claim gas yield for address(0), then all the gas yield (user gas spent on contract PoolDeployer) is lost.

Recommendation

change address(0) to address(this)

M2 –Winning users can lose their reward and prize depends on winning tickets and max count configuration

Description

In the current implementation,

whether user can claim the reward with their winning tickets depends on winning tickets setting and maxPrizeCount setting

```
    function claimPrizesForRound(uint256 roundToClaim) external {
            require(roundStart[roundToClaim] + MAX_ROUND_TIME >=
  block.timestamp, "ICT");
            require(winningTickets[roundToClaim][0].length > 0, "NWT");
            Round memory round = entered[msg.sender][roundToClaim];
            require(round.ticketEnd > round.ticketStart, "NTE");
            uint256 maxPrizeCount_ = maxPrizeCount;
            for (uint256 i = 0; i < maxPrizeCount_; i++) {
                Prize memory prize = prizes[roundToClaim][i];
                uint256[] memory winningTicketsRoundPrize =
  winningTickets[roundToClaim][i];
                for (uint256 j = 0; j < winningTicketsRoundPrize.length; j++) {
                    uint256 winningTicket = winningTicketsRoundPrize[j];
                    if (round.ticketStart <= winningTicket && round.ticketEnd >
  winningTicket) {
                      _claimPrize(prize, msg.sender, winningTicket);
                    }
                }
            }
            entered[msg.sender][roundToClaim] = Round(0, 0, roundToClaim); //
  Clear user's tickets for the round
            emit CheckedPrizesForRound(msg.sender, roundToClaim);
        }
```

suppose user's winningTickets[roundToClaim] is in number 20 to claim the prize,

but maxPrizeCount, which applies globally, is 10, the i will never be 20 and the for loop iternation ends when i < maxPrizeCount_ early, meaing the data

```
winningTickets[roundToClaim][20];
```

will not be accessed. then the enter's round to claim is reset

```
entered[msg.sender][roundToClaim] = Round(0, 0, roundToClaim); // Clear
user's tickets for the round
```

User lose their reward and prize. Or we consider another case, suppose initially maxPrizeCount set by admin is 15, It could also be the case when prizes[roundToClaim] is 12 when user has winning ticket, then admin change and reduce the maxPrizeCount to 10, then as we can see, the nested for ends early and the i will not be 12 for use to claim the prize.

```
Prize memory prize = prizes[roundToClaim][i];
uint256[] memory winningTicketsRoundPrize = winningTickets[roundToClaim]
[i];
```

Recommendation

Consider adding a restriction that winningTickets[roundToClaim] length and prizes[roundToClaim] length should always below the maxPrizeCount and restrict the winning ticket number.

Protocol Response:

Protocol will consider add a check that maxPrizeCount never decreases. Realistically Admin not be decreasing the maxPrizeCount.

L1 – Yield cut is not intialized in cfmm factory

Description

```
contract ThrusterFactory is IThrusterFactory, ThrusterGas {

    uint256 public yieldCut;
```

In ThrusterFactory contract, the admin can set the yield cut percentage, but the default yieldCut is not initialized.

```
    function setYieldCut(uint256 _yieldCut) external {
          require(msg.sender == yieldToSetter, "ThrusterFactory: FORBIDDEN");
          require(_yieldCut < 6 && _yieldCut > 0, "ThrusterFactory:
  INVALID_YIELD_CUT"); // Yield cut is 16-50% of the LP yield
          yieldCut = _yieldCut;
          emit SetYieldCut(_yieldCut);
      }
```

as the require statement's check, the yield cut should not be 0, but the default yieldCut is never initialized, if the admin intends to collect fee, but the yield cut is not set, then when calling _mintYieldcut

```
uint256 denominator =
rootK.mul(IThrusterFactory(factory).yieldCut()).add(rootKLast);
uint256 liquidity = numerator / denominator;
```

then protocol lose a part of the liquidity fee in ThrusterPair.sol contract.

Recommendation

Initialize the yieldCut parameter in ThrusterFactory constructor

L2 – Entropy provider address in Thruster treasury should be changeable

Description

In the current implementation, once the entropy provider address is set in Thruster treausry contract, the entropyProvider address cannot be changed.

the entropy provider contract cannot be changed

```
    entropy = IEntropy(_entropy);
    entropyProvider = _entropyProvider;
```

but if we take a look at the entropy implementation on pyth network size,

There can be new provider added

New address can become providers, while old providers can adjust the fee. In the case when the old provider is deprecated or the old provider set the fee absurdly high, the protocol should be able to change the provider supported.

Recommendation

add a function to set the entropy provider in ThrusterTreasury contract.

L3 – USDB / WETHB contract can be changed in mainnet

Description

Through the codebase, the address for BLAST, USDB, WETHB are hardcoded

```
contract ThrusterYield is IThrusterYield {

    IBlast public constant BLAST =
IBlast(0x4300000000000000000000000000000000000002);
    IERC20Rebasing public constant USDB =
IERC20Rebasing(0x4200000000000000000000000000000000000022);
    IERC20Rebasing public constant WETHB =
IERC20Rebasing(0x4200000000000000000000000000000000000023);
```

but these address are just testnet address, and the mainnet address for these contract is not set and known yet.

https://docs.blast.io/building/bridged-token-addresses

and

https://docs.blast.io/building/contracts

Recommendation

The recommendation is pass these address in the constructor instead of hardcode them using testnet address for mainnet deployment.

L4 – Incorrect chain id check in NonfungibleTokenPositionDescriptor.sol implementation

Description

The blast mainnet chain id is to be set, the the chain id of blast mainnet is very unlikely be 1

```
     function flipRatio(address token0, address token1, uint256 chainId) public
    view returns (bool) {
            return tokenRatioPriority(token0, chainId) >
    tokenRatioPriority(token1, chainId);
        }

        function tokenRatioPriority(address token, uint256 chainId) public view
    returns (int256) {
            if (token == WETH9) {
                return TokenRatioSortOrder.DENOMINATOR;
            }
            if (chainId == 1) {
                if (token == USDB) {
                    return TokenRatioSortOrder.NUMERATOR_MOST;
                } else {
                    return 0;
                }
            }
            return 0;
        }
```

Recommendation

It is recommendation to remove the check

```
if (chainId == 1)
```

R1 - Yield claim mechanism can be more efficient

Description

In the current implementation, the admin has to claim the gas yield and WETH / USDB yield for each Pool

but consider a lot of user creates a lot of trading pairs and 10000 trading pairs exists, for every pair address, admin needs to starts one transaction to claim the yield.

the admin needs to starts 10000 transaction to claim the yield for all contract

Recommendation

There is more efficient way to claim the yield in batch, the protocol can set the manager to a smart contract address and implement batch claim feature to claim yield from multiple contract. The much more efficient way to claim the yield is let user claim the yield very time when a swap / mint / burn happens. Then protocol does not have to claim the yield for 10000 times.

R2 – Consider set a yield governor to claim the yield for protocol to reduce the contract size

Description

Based on

https://github.com/ThrusterX/thruster-protocol/blob/main/ThrusterAudits_Changelog.pdf

the protocol is concern about the deployment contract size for ThrusterPool, one of the function that protocol added is the function "claimYieldAll"

```
function claimYieldAll(address _recipient, uint256 _amountETH, uint256
_amountWETH, uint256 _amountUSDB)
        external
        override
        onlyFactoryOwner
        returns (uint256 amountETH, uint256 amountWETH, uint256 amountUSDB,
uint256 amountGas)
    {
        amountETH = IBlast(BLAST).claimYield(address(this), _recipient,
_amountETH);
        amountWETH = IERC20Rebasing(WETHB).claim(_recipient, _amountWETH);
        amountUSDB = IERC20Rebasing(USDB).claim(_recipient, _amountUSDB);
        amountGas = IBlast(BLAST).claimMaxGas(address(this), _recipient);
    }
```

This function does increase the contract size.

Recommendation

To further reduce the contract size, the protocol can set the governor of the contract, then governor can claim yield on behalf of contract seperately

```
 /**
     * @notice Configures the governor for the contract that calls this
function
     */
    function configureGovernor(address _governor) external {
        require(isAuthorized(msg.sender), "not authorized to configure
contract");
        governorMap[msg.sender] = _governor;
    }
```

and then the function claimYieldAll can be removed.

R3 – revealRandomNumber function should have internal modifier instead of public modifier

Description

In the current implementation, the function revealRandomNumber has public modifier and only owner can call this function

Howevever, this function is not desigend to be called directly, it is only designed to be called in the for loop setWinningTicket, as the comments in entropy contract from Pyth said

```
 // Note that this function can only be called once per in-flight request.
Calling this function deletes the stored
    // request information (so that the contract doesn't use a linear
amount of storage in the number of requests).
    // If you need to use the returned random number more than once, you
are responsible for storing it.
    //
    // This function must be called by the same `msg.sender` that
originally requested the random number. This check
    // prevents denial-of-service attacks where another actor front-runs
the requester's reveal transaction.
    function reveal(
        address provider,
        uint64 sequenceNumber,
        bytes32 userRandomness,
        bytes32 providerRevelation
    ) public override returns (bytes32 randomNumber) {
```

calling reveal should store the returned results and the result cannot be revealed several times,

if the admin calls revealNumber by accidents, the protocol lose the request fee from prior request and have to make an additional random request to make up for the deleted one.

Recommendation

change the revealNumber access modifier from public to internal.

R4 – User claim reward process can be more gas efficient

Description

When user claim the price, the user has to pay the gas to run massive nested for loop

```
for (uint256 i = 0; i < maxPrizeCount_; i++) {
    Prize memory prize = prizes[roundToClaim][i];
    uint256[] memory winningTicketsRoundPrize =
winningTickets[roundToClaim][i];
    for (uint256 j = 0; j < winningTicketsRoundPrize.length; j++) {
        uint256 winningTicket = winningTicketsRoundPrize[j];
        if (round.ticketStart <= winningTicket && round.ticketEnd >
winningTicket) {
            _claimPrize(prize, msg.sender, winningTicket);
        }
    }
}
```

If the user's winning round and winning ticket number is large enough, the transaction can run out of gas. Also noted that if there are multiple winners, the protocol must deposit fund for every winners every time, which is not a efficient way to accomolating a lot of users.

```
   function setPrize(uint256 _round, uint64 _prizeIndex, uint256
_amountWETH, uint256 _amountUSDB, uint64 _numWinners)
        external
        onlyOwner
    {
        require(_round >= currentRound, "ICR");
        require(_prizeIndex < maxPrizeCount, "IPC");
        depositPrize(msg.sender, _amountWETH, _amountUSDB);
        prizes[_round][_prizeIndex] = Prize(_amountWETH, _amountUSDB,
_numWinners, _prizeIndex, uint64(_round));
    }

    /**
     * Deposits the prize amounts determined in setPrize
     *
     * @param _from - The address who should deposit the prize
     * @param _amountWETH - The amount of WETH
     * @param _amountUSDB - The amount of USDB
     */
    function depositPrize(address _from, uint256 _amountWETH, uint256
_amountUSDB) internal {
        WETH.transferFrom(_from, address(this), _amountWETH);
        USDB.transferFrom(_from, address(this), _amountUSDB);
        emit DepositedPrizes(_amountWETH, _amountUSDB);
    }
```

Recommendation

Consider a alternative approach, admin can set a merkle tree root and user can offer proof to claim the reward, the validation schema should be

```
bytes32 node = keccak256(abi.encodePacked(
    msg.sender,
    round_number,
    winning_tickets, // an array of winning tickets
    usdb_claim_amount,
    weth_claim_amount
    ));
require(MerkleProof.verify(_proof, root, node), "IP");
```

then user does not have to pay the gas to run the nested for loop.

Additional Consideration and notes

Make sure the points admin works as intended

At the time of audits,there is lack of documentation for how the points assignment for contracts works

https://docs.blast.io/about-blast

It is highly recommendated to consistently look for updates about the contract points distribution to make sure
protocold does not lose airdrop / yield if the points distribution is related to yield

Make sure the added gauge logic works as intended

The gauege contract is not in the scope of audit, While adding the gauge logic does not break the math invariant,

it is recommended to make sure that calling

```
    if (address(gauge) != address(0)) {
        IThrusterGauge(gauge).checkpoint(cache.blockTimestamp);
    }
```

and

```
if (gauge != address(0)) {
    IThrusterGauge(gauge).cross(step.tickNext, zeroForOne);
}
```

works as intended.

Consider add more tesing for ThrusterTreasury.

It is recommended to add more unit test and integration testing for ThrusterTreasury, especially in the area of reward claiming and distribution.

Additional resources for blast contract

In case the protocol is interested, here is a list of contract blast integrated with:

the blast contract:

https://github.com/blast-io/blast/blob/master/blast-optimism/packages/contracts-bedrock/src/L2/Blast.sol#L211

and the gas contraact:

https://github.com/blast-io/blast/blob/master/blast-optimism/packages/contracts-bedrock/src/L2/Gas.sol

the precompile contract that handles native ETH yield

https://github.com/blast-io/blast/blob/78003ff2bd78cb07d3c830a24929d7670b2a1f4d/blast-geth/core/vm/contracts.go#L1209