code4rena

Gondi

Smart Contract Security Assessment

Version 1.0

Audit dates: Jan 15 — Jan 21, 2025

Audited by: SpicyMeatball
oakcobalt

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 Gondi

GONDI is a decentralized peer-to-peer non-custodial NFT lending protocol that aims to offer the most flexible and capital-efficient primitive.

2.2 Scope

Repository	pixeldaogg/florida-contracts
Commit Hash	758308774b92a8f9bdf64e13538abb3d11372734

2.3 Audit Timeline

DATE	EVENT
Jan 15, 2025	Audit start
Jan 21, 2025	Audit end
Feb 04, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	4
Medium Risk	7
Low Risk	3
Informational	4
Total Issues	18

3. Findings Summary

ID	DESCRIPTION	STATUS
H-1	executeSellWithETH will lock NFTs during batch transactions, risks of NFT theft	Resolved
H-2	Buyer's address is not saved to transient storage in executeSell	Resolved

H-3	Incorrect router approval will DOS swap and sell flow	Resolved
H-4	Buyer's remaining Weth will not be refunded	Resolved
M-1	Bundler is incompatible with USDT	Resolved
M-2	Users have full control over bundler's approvals	Resolved
M-3	ERC20 asset approval is not reset during executeSell	Resolved
M-4	ERC20 transfers not checking return value	Resolved
M-5	In the executeOperation flow, remaining funds may not be returned	Resolved
M-6	executeSellWithETH always revert when the collateral NFT is wrapped punk sold in bundler	Resolved
M-7	Remaining principal not transferred to the borrower, risks of exploits	Resolved
L-1	Invalid whitelisted currency check	Resolved
L-2	Multiple flows have insufficient input validations	Resolved
L-3	Incorrect order typehash	Resolved
I-1	Some token transfer implementation not compatible with ERC20 tokens that revert on zero transfer	Resolved
I-2	Remove unused imports	Resolved
I-3	_paybackRemainingWeth function is not used in the bundler	Resolved
I-4	Unnecessary code - order.taker is checked twice	Resolved

4. Findings

4.1 High Risk

A total of 4 high risk findings were identified.

[H-1] executeSellWithETH will lock NFTs during batch transactions, risks of NFT theft

Severity: High	Status: Resolved

Target

- PurchaseBundler.sol#L218

Severity:

- Impact: High
- Likelihood: Medium

Description: executeSell, Sell, and exectueSellWithETH all allow batch transactions (bytes[] executionData). A buyer can buy multiple collateral NFTs in one tx.

However, exectueSellWithETH doesn't handle batch transactions correctly. The vulnerable case is selling NFT in the local marketplace (TradeMarketPlace). (exectueSellWithETH -> _sell -> afterNFTTransfer -> _sellReleasedCollateral -> _executeOrder)

In _executeOrder, `taker` will be address(this), which means the NFT token is always transferred from the borrower to the PurchaseBundler.

```
    function _executeOrder(Order memory order, address taker) internal {
...
        if (order.isAsk) {
            _transferERC20(currency, taker, order.maker, order.price);
|>         ERC721(order.collection).safeTransferFrom(order.maker, taker,
order.tokenId);
        } else {
```

- TradeMarketplace.sol#L46

Back in `exectueSellWithETH`, we see only one token `uint256 tokenId` will be transferred to the buyer/caller. All other tokens will stay in the bundler contract. The tx succeeds.

```
    function executeSellWithETH(
        bytes calldata wethPrincipalSwapData,
        ERC20 principal,
        ERC721 collection,
        uint256 tokenId,
        bytes[] calldata executionData
    ) external payable _storeMsgSender {
        _weth.deposit{value: msg.value}();
        _swapWETH(wethPrincipalSwapData);
        _sell(executionData);
        if (collection.ownerOf(tokenId) != msg.sender) {
            //// @dev we need to do this because Seaport private listings
transfer directly to recipient
            //// if the recipient is not the caller, this will fail and
revert.
|>          collection.safeTransferFrom(address(this), msg.sender,
tokenId);
        }
        principal.safeTransfer(msg.sender,
principal.balanceOf(address(this)));
    }
```

- [PurchaseBundler.sol#L218]()

In addition, due to another vulnerability in `executeSell()` where `bytes[] calldata executionData` input is not validated. A malicious user can steal any locked NFT tokens by passing empty executionData when back running `exectueSellWithETH` tx to steal the locked NFTs.

See added unit test with modified helpers:

```
    function testExecuteWETHLoanSellWithETH_batch_exploits() public {
        (
            uint256 loanId,
            IMultiSourceLoan.Loan memory loan
        ) = _getInitialLoanWithPrincipal(SampleToken(_weth));
        collateralCollection.mint(_borrower, 2);
        (
            uint256 loanId2,
            IMultiSourceLoan.Loan memory loan2
        ) = _getInitialLoanWithPrincipal_w_tokenId(SampleToken(_weth),
2);
        vm.warp(1 days);

        uint256 owed = loan.getTotalOwed(block.timestamp);
        uint256 profit = 100;
        uint256 price = owed + profit;
        ERC721 collateral = ERC721(loan.nftCollateralAddress);
        uint256 tokenId = loan.nftCollateralTokenId;
        uint256 tokenId2 = loan2.nftCollateralTokenId;
        ERC20 principal = ERC20(loan.principalAddress);

        bytes memory swapData = "";
        bytes[] memory repaymentData = new bytes[](2);
        repaymentData[0] = abi.encodeWithSelector(
            IMultiSourceLoan.repayLoan.selector,

_getSampleExecutionDataForLoanSaleInPurchaseBundler_w_tokenId(
                loanId,
                loan,
                address(_purchaseBundler),
                price,
                _purchaseBundler
            )
        );
        repaymentData[1] = abi.encodeWithSelector(
            IMultiSourceLoan.repayLoan.selector,

_getSampleExecutionDataForLoanSaleInPurchaseBundler_w_tokenId(
                loanId2,
                loan2,
                address(_purchaseBundler),
                price,
                _purchaseBundler
            )
        );
        vm.deal(_buyer, price * 2);
        deal(_weth, _borrower, 0);
        deal(_weth, address(_purchaseBundler), 0);
        vm.startPrank(_borrower);
        collateral.setApprovalForAll(address(_purchaseBundler), true);
        principal.approve(address(_msLoan), 2 * price);
        vm.stopPrank();
        vm.prank(_buyer);
        _purchaseBundler.executeSellWithETH{value: price * 2}(
            swapData,
            principal,
            collateral,
            tokenId,
            repaymentData
        );
        assertEq(principal.balanceOf(_borrower), 2 * profit);
        assertEq(principal.balanceOf(address(_purchaseBundler)), 0);
        assertEq(collateral.ownerOf(tokenId), _buyer);
        assertEq(collateral.ownerOf(tokenId2),
address(_purchaseBundler)); // tokenId2 is locked in bundler
        vm.prank(userB);
        ERC721[] memory _collections = new ERC721[](1);
        _collections[0] = collateral;
        uint256[] memory _tokenIds = new uint256[](1);
        _tokenIds[0] = tokenId2;
        _purchaseBundler.executeSell(
            new ERC20[](0),
            new uint256[](0),
            _collections,
            _tokenIds,
            address(0x1),
            new bytes[](0)
        );
        assertEq(collateral.ownerOf(tokenId2), userB); // attacker get
the locked tokenId2
    }
```

Recommendation: Allow check and transfer all collections and tokenIds in the batch data:

```
        for (uint256 i = 0; i < collections.length; ++i) {
            if (collections[i].ownerOf(tokenIds[i]) != msg.sender) {
                //// @dev we need to do this because Seaport private
listings transfer directly to recipient
                //// if the recipient is not the caller, this will fail
and revert.
                collections[i].safeTransferFrom(address(this),
msg.sender, tokenIds[i]);
            }
        }
```

Gondi: Fixed with PR-447

Zenith: Verified. Removed batch tx capability for executeSellWithETH.

[H-2] Buyer's address is not saved to transient storage in executeSell

Severity: High	Status: Resolved

Target

- PurchaseBundler.sol#L168-L200
- PurchaseBundler.sol#L537

Severity:

- Impact: High
- Likelihood: Medium

Description: When a user attempts to buy a collateral NFT from the borrower via `executeSellWithETH` or `executeSellWithLoan`, their address is stored in transient storage for further use:

```
modifier _storeMsgSender() {
    address msgSender = msg.sender;
    assembly {
        tstore(_MSG_SENDER_TOFFSET, msgSender)
    }
    _;
    assembly {
        tstore(_MSG_SENDER_TOFFSET, 0)
    }
}

function _msgSender() private view returns (address msgSender) {
    assembly {
        msgSender := tload(_MSG_SENDER_TOFFSET)
    }
    if (msgSender == address(0)) {
        msgSender = msg.sender;
    }
}
```

However, in the `executeSell` function, `_storeMsgSender` is not called. This can lead to unintended consequences if the `afterNFTTransfer` callback goes through the `_sellReleasedCollateral` path, where the NFT seller and buyer are matched in the bundler's internal marketplace:

```
    function _sellReleasedCollateral(IMultiSourceLoan.Loan memory loan,
IReservoir.ExecutionInfo memory executionInfo)
        private
        returns (bool success)
    {
        ITradeMarketplace.Order memory order =
abi.decode(executionInfo.data, (ITradeMarketplace.Order));
        ERC721 collateral = ERC721(order.collection);
        ERC20 principal = ERC20(loan.principalAddress);
        uint256 tokenId = order.tokenId;
>>      address taker = _msgSender();

        require(address(principal) == order.currency, "Principal
mismatch");

        _executeOrder(order, address(this));

        if (_isPunkWrapper(collateral)) {
            _unwrapPunk(collateral, tokenId);
>>          _punkMarket.transferPunk(taker, tokenId);
        }
        return true;
    }
```

Since the buyer's address is not stored in transient storage, `_msgSender` will return the `MultiSourceLoan` contract address, as it is the `msg.sender` in this context. This will result in the collateral being sent to the contract address instead of the buyer.

Several impacts may occur:

- if the collateral is an ERC721 token, the call will revert here:

```
        for (uint256 i = 0; i < collections.length; ++i) {
            if (collections[i].ownerOf(tokenIds[i]) != buyer) {
                //// @dev we need to do this because Seaport private
listings transfer directly to recipient
                //// if the recipient is not the caller, this will fail
and revert.
                collections[i].safeTransferFrom(address(this), buyer,
tokenIds[i]);
            }
```

- If the collateral is a CryptoPunk token, the call will not revert, as CryptoPunks do not have an `ownerOf` function. So the buyer has to trust the bundler to send the token to them, but it will instead be sent to another address, resulting in token loss.

Recommendation: Add the `_storeMsgSender` modifier to the `executeSell` function. A potential complication arises because this function is also called from within `executeOperation`:

```
    function executeOperation(
        address[] calldata assets,
        uint256[] calldata amounts,
        uint256[] calldata premiums,
        address,
        bytes calldata params
    ) external override(IAaveFlashLoanReceiver) returns (bool) {
        (ExecuteSellWithLoanArgs memory args) = abi.decode(params,
(ExecuteSellWithLoanArgs));
        this.executeSell(
```

This can be resolved by creating two versions of `executeSell`:

- one for normal user interactions that includes the _storeMsgSender modifier;
- one for flashloan operations that does not require it.

Gondi: Fixed in PR-449

Zenith: Verified. The buyer's address is stored in transient memory in `executeSell`. If an address is already saved, it won't be overwritten, ensuring no conflict with the `executeOperation` function.

[H-3] Incorrect router approval will DOS swap and sell flow

Severity: High	Status: Resolved

Target

- PurchaseBundler.sol#L138
- PurchaseBundler.sol#L150

Severity:

- Impact: Medium
- Likelihood: High

Description: Current token approval to uniswap universal router is set directly on ERC20 token contracts. This is vulnerable because uniswap universal router only checks approvals and transfers through PERMIT2.

The swap and sell flow(executeSellWithETH) will be DOSsed. (executeSellWithETH -> _swapWETH -> universalRouter::execute -> ... -> permit2TransferFrom).

Currently, token approval is set in the constructor and approveForSwap in ERC20.

```
//src/lib/callbacks/PurchaseBundler.sol
    constructor(
...
    ) WithProtocolFee(tx.origin, minWaitTime, protocolFee)
TradeMarketplace(name) {
...
|>      ERC20(address(_weth)).approve(address(_uniswapRouter),
type(uint256).max);
    }
```

- PurchaseBundler.sol#L138

```
    function approveForSwap(address currency) external {
        _currencyManager.isWhitelisted(currency);
|>      ERC20(currency).approve(address(_uniswapRouter),
type(uint256).max);
    }
```

- PurchaseBundler.sol#L150

But universal router only uses `PERMIT2.transferFrom` for asset `transferFrom`. For example,

In uniswap v3 exact output swap, universalRouter will invoke v3SwapExactOutput. The asset transfer is done in uniswapV3SwapCallback which calls PERMIT2.transferFrom(from, to, amount, token) . Allowance set on PERMIT2 is checked here.

```
    function permit2TransferFrom(address token, address from, address to,
uint160 amount) internal {
        PERMIT2.transferFrom(from, to, amount, token);
    }
```

- Permit2Payments.sol#L21

Recommendation: First, approve PERMIT2 on WETH contract in constructor. Then set weth approval to uniswapRouter in Permit2 atomically in _swapWETH. For example,

```
    function _swapWETH(bytes calldata swapData) private {
        if (swapData.length == 0) return;
+       PERMIT2.approve(address(_weth), _uniswapRouter, type(uint160).max
,0);
```

Gondi: Fixed with PR-452

Zenith: Verified. Approval is revised based on Permit2.

[H-4] Buyer's remaining Weth will not be refunded

Severity: High	Status: Resolved

Target

- PurchaseBundler.sol::executeSellWithETH

Severity:

- Impact: Medium
- Likelihood: High

Description: executeSellWithEth allows a buyer to supply ETH and swap into the principal token required to purchase the collateral NFT.

The issue is only the remaining principal tokens are refunded. If the swap doesn't use all the ETH supplied by the buyer, the remaining WETH will not be returned. This is common with exactOutput swap where the amount_in changes dynamically.

Vulnerable case: msg.value > WETH used in swap We see msg.value is converted into WETH. Uniswap router will only attempt to transfer the calculated amount in assets. Only the principal token amount is checked and refunded. The remaining WETH is left in the contract.

```
    function executeSellWithETH(
        bytes calldata wethPrincipalSwapData,
        ERC20 principal,
        ERC721 collection,
        uint256 tokenId,
        bytes[] calldata executionData
    ) external payable _storeMsgSender {
|>     _weth.deposit{value: msg.value}();
        _swapWETH(wethPrincipalSwapData);
        _sell(executionData);
        if (collection.ownerOf(tokenId) != msg.sender) {
            //// @dev we need to do this because Seaport private listings
transfer directly to recipient
            //// if the recipient is not the caller, this will fail and
revert.
            collection.safeTransferFrom(address(this), msg.sender,
tokenId);
        }
|>      principal.safeTransfer(msg.sender,
principal.balanceOf(address(this)));
```

```
    }
```

- PurchaseBundler.sol#L220

In addition, because of a separate vulnerability in `executeSell` where input validations are not checked, a malicious user can backrun someone's `executeSellWithETH` to steal the locked WETH. For example,

The malicious actor can call `executeSell` with empty `executionData`, 0 as currencyAmounts[0], empty `collections` and weth as currencies[0]. _paybackRemaining will transfer the previous user's locked weth to the caller.

Recommendation:

```
    function executeSellWithETH(
...
        principal.safeTransfer(msg.sender,
principal.balanceOf(address(this)));
+      _paybackRemainingWeth();
    }
```

Gondi: Fixed with PR-459

Zenith: Verified. Weth refund step is added.

4.2 Medium Risk

A total of 7 medium risk findings were identified.

[M-1] Bundler is incompatible with USDT

Severity: Medium	Status: Resolved

Target

- PurchaseBundler.sol#L150
- PurchaseBundler.sol#L182
- PurchaseBundler.sol#L261

Severity:

- Impact: Medium
- Likelihood: Medium

Description: The `PurchaseBundler` contract casts the asset address to the `ERC20` type from the Solmate package, which follows a standard EIP-20 implementation. However, if the asset is a USDT token, the `approve` transaction will revert because the compiler expects a `bool` return value, whereas USDT's `approve` function does not return anything. This mismatch causes a decoding error, leading to transaction failure.

Recommendation: It is recommended to use `safeApprove` for all assets.

Gondi: Fixed in PR-466

Zenith: Verified.

[M-2] Users have full control over bundler's approvals

Severity: Medium	Status: Resolved

Target

- PurchaseBundler.sol#L182

Severity:

- Impact: Medium
- Likelihood: Medium

Description: It is possible for users to approve any amount of tokens to an arbitrary address on behalf of the bundler's contract in the `executeSell` function:

```
    function executeSell(
        ERC20[] calldata currencies,
        uint256[] calldata currencyAmounts,
        ERC721[] calldata collections,
        uint256[] calldata tokenIds,
        address marketPlace,
        bytes[] calldata executionData
    ) external {
        address buyer = _msgSender();
        for (uint256 i = 0; i < currencies.length; ++i) {
            uint256 balance = currencies[i].balanceOf(address(this));
            if (currencyAmounts[i] > balance) {
                currencies[i].safeTransferFrom(buyer, address(this),
currencyAmounts[i] - balance);
            }
>>          currencies[i].approve(marketPlace, currencyAmounts[i]);
```

The only prerequisite for this action is that the caller must send tokens to the contract. Since the tokens are refunded to the caller anyway, there is no direct risk to the caller.

There are several potential issues:

- users can set the Uniswap router's allowance to zero for an asset by specifying its address in the `marketPlace` argument;
- another vulnerability is that users can approve tokens to the Uniswap router that are not whitelisted in the currency manager.

Recommendation: Maintain a `marketPlace` whitelist or, at the very least, disallow specifying `_uniswapRouter` as the `marketPlace` in `executeSell`.

Gondi: Fixed in [PR-448]

Zenith: Verified. The `marketPlace` parameter is validated in `executeSell`.

[M-3] ERC20 asset approval is not reset during executeSell

Severity: Medium	Status: Resolved

Target

- PurchaseBundler.sol#L182

Severity:

- Impact: Medium
- Likelihood: Medium

Description: USDC and USDT tokens require that their allowance to a given address be set to 0 before changing it to another value via an `approve` call. This behavior can cause issues in the `executeSell` function:

```
    function executeSell(
        ERC20[] calldata currencies,
        uint256[] calldata currencyAmounts,
        ERC721[] calldata collections,
        uint256[] calldata tokenIds,
        address marketPlace,
        bytes[] calldata executionData
    ) external {
        address buyer = _msgSender();
        for (uint256 i = 0; i < currencies.length; ++i) {
            uint256 balance = currencies[i].balanceOf(address(this));
            if (currencyAmounts[i] > balance) {
                currencies[i].safeTransferFrom(buyer, address(this),
currencyAmounts[i] - balance);
            }
>>          currencies[i].approve(marketPlace, currencyAmounts[i]);
        }
```

If the full allowance for these tokens is not consumed during the function call, subsequent `executeSell` calls that attempt to approve stablecoins for an address with a non-zero allowance will revert.

This issue can also be exploited maliciously. For example, a user could call executeSell with zero-length executionData and USDT/USDC tokens as currencies. The tokens would be approved for an arbitrary address, but since no execution occurs, the assets would be returned to the sender while leaving the allowance non-zero.

Recommendation: Reset the allowance for the marketplace to zero at the end of the `executeSell` function to prevent this issue.

Gondi: Fixed in PR-451

Zenith: Verified. ERC20 approvals are reset after a sale.

[M-4] ERC20 transfers not checking return value

Severity: Medium	Status: Resolved

Target

- TradeMarketplace.sol#L108-L114

Severity:

- Impact: High
- Likelihood: Low

Description: When interacting with ERC20 tokens, the `TradeMarketplace` contract does not check whether the token transfer was successful:

```
  function _transferERC20(ERC20 token, address from, address to,
uint256 amount) private {
    if (from == address(this)) {
>>      token.transfer(to, amount);
    } else {
>>      token.transferFrom(from, to, amount);
    }
  }
```

Some ERC20 tokens do not revert on errors. Instead, they return a boolean value indicating whether the transaction was successful.

Recommendation: Consider using SafeERC20 library.

Gondi: Fixed in PR-456

Zenith: Verified. Solmate `SafeTransferLib` is used to ensure the ERC20 transfer is successful.

[M-5] In the executeOperation flow, remaining funds may not be returned

Severity: Medium

Status: Resolved

Target

- PurchaseBundler.sol#L480
- PurchaseBundler.sol#L258

Severity:

- Impact: Medium
- Likelihood: Medium

Description: A buyer can take out flashloan to buy a collateral nft and then atomically take on a loan with the nft. (executeSellWithLoan -> executeOperation -> executeSell -> _paybackRemaining)

The remaining funds handling in the complex flow is vulnerable: (1) _paybackRemaining In executeSell step, if any assets remained after the nft purchase, _paybackRemaining is supposed to send it to the buyer. But _paybackRemaining uses msg.sender instead of the wrapped _msgSender().

Due to `executeSell` is invoked in a self call (`this.executeSell`), msg.sender will be address(this). Any remaining methods will only be self-transferred.

```
    function _paybackRemaining(ERC20 currency) private {
        uint256 remaining = currency.balanceOf(address(this));
|>     currency.safeTransfer(msg.sender, remaining);
    }
```

- PurchaseBundler.sol#L480

(2) Missing `asset.balanceOf(address(this)) > owed` handling in `executeOperation` When `asset.balanceOf(address(this)) > owed`, this contract holds more funds than the required flashloan repayment. This might happen due to remaining funds from (1) and also the principal received from the new loan(when new principal is greater than NFT offer price). We see `asset.balanceOf(address(this)) > owed` case is not handled in `executeOperation`.

```
    function executeOperation(
...
        for (uint256 i = 0; i < assets.length; i++) {
            address _buyer = _msgSender();
            uint256 owed = amounts[i] + premiums[i];
            ERC20 asset = ERC20(assets[i]);
            if (asset.balanceOf(address(this)) < owed) {
                asset.safeTransferFrom(_buyer, address(this), owed -
asset.balanceOf(address(this)));
            }
            asset.approve(address(args.borrowArgs.pool), owed);
        }
```

- PurchaseBundler.sol#L258

Both cases may cause surplus funds not returned to the buyer.

Recommendation: (1) Consider changing msg.sender into _msgSender in _paybackRemaining. (2) Consider adding a check condition if `asset.balanceOf(address(this)) > owed` , transfer the extra funds to _msgSender

Gondi: Fixed with PR-461

Zenith: Verified. Check for remaining funds is added.

[M-6] executeSellWithETH always revert when the collateral NFT is wrapped punk sold in bundler

Severity: Medium	Status: Resolved

Target

- PurchaseBundler.sol#L543-L545

Severity:

- Impact: Medium
- Likelihood: Medium

Description: `executeSellWithETH` is compatible with `_sellReleasedCollateral`, allowing a buyer to buy collateral NFT from the bundler contract(TradeMarketPlace) by supplying ETH.

The problem is `executeSellWithETH` doesn't handle the case where the collateral NFT sold is wrapped punk correctly.

Vulnerable flow: PurchaseBundler::executeSellWithETH -> ... -> _sellReleasedCollateral -> _isPunkWrapper(collateral) == true We see in `_sellReleasedCollateral`, the bundler contract will hold the wrapped punk , then unwrap it and transfer the raw cryptoPunk NFT directly to the buyer.

```
//florida-contracts/src/lib/callbacks/PurchaseBundler.sol

    function _sellReleasedCollateral(IMultiSourceLoan.Loan memory loan,
IReservoir.ExecutionInfo memory executionInfo)
        private
        returns (bool success)
    {
        ITradeMarketplace.Order memory order =
abi.decode(executionInfo.data, (ITradeMarketplace.Order));
        ERC721 collateral = ERC721(order.collection);
        ERC20 principal = ERC20(loan.principalAddress);
        uint256 tokenId = order.tokenId;
        address taker = _msgSender();

        require(address(principal) == order.currency, "Principal
mismatch");

        _executeOrder(order, address(this));

       if (_isPunkWrapper(collateral)) {
            _unwrapPunk(collateral, tokenId);
|>         _punkMarket.transferPunk(taker, tokenId); //@audit-info note:
this transfer the unwrapped punk to taker(buyer)
        }
        return true;
    }
```

- PurchaseBundler.sol#L545

Back in `executeSellWithETH`, after the _sell flow, collection.ownerOf is checked and transferred again. However, due to wrappedPunk(tokenId) is already burned, `collection.ownerOf(tokenId) != msg.sender` will evaluate to true, and the bundler will try to transfer a burned tokenId, reverting the entire tx.

```
    function executeSellWithETH(
        bytes calldata wethPrincipalSwapData,
        ERC20 principal,
        ERC721 collection,
        uint256 tokenId,
        bytes[] calldata executionData
    ) external payable _storeMsgSender {
        _weth.deposit{value: msg.value}();
        _swapWETH(wethPrincipalSwapData);
        _sell(executionData);

        if (collection.ownerOf(tokenId) != msg.sender) {
            //// @dev we need to do this because Seaport private listings
transfer directly to recipient
            //// if the recipient is not the caller, this will fail and
revert.
|>          collection.safeTransferFrom(address(this), msg.sender,
tokenId);
        }
        principal.safeTransfer(msg.sender,
principal.balanceOf(address(this)));
    }
```

In addition, the executeSell flow(-> _sellReleasedCollateral) is also incompatible when selling wrapped punk for the same reason.

Recommendation: In `_sellReleasedCollateral`, consider removing the `if (_isPunkWrapper(collateral)) {` branch so that the wrapped version will be checked to be owned by the buyer.

Gondi: PR-463

Zenith: Verified. Added punk collection handling `_givebackNFTOrPunk`.

[M-7] Remaining principal not transferred to the borrower, risks of exploits

Severity: Medium	Status: Resolved

Target

- PurchaseBundler.sol#L446

Severity:

- Impact: High
- Likelihood: Low

Description: A borrower can buy an NFT as collateral with loan principal(Weth) through `buy`. The total principal minus fee will be transferred to PurchaseBundler. It's possible an NFT token id can be offered at a cheaper price in a market than principal.

The vulnerability is that if there is any surplus principal(`remainingBalance`), the principal is not transferred back to the borrower, but to the caller(`msg.sender`). This also creates risks of exploits because one borrower's surplus principal can be socialized to buy another borrower's collaterals.

Vulnerable case: msg.sender != borrower Because an operator or anyone can also execute `buy` to `emitLoan` on a borrower's behalf with their signature, and if there are remaining principals left, funds are transferred to the wrong address.

```
//src/lib/callbacks/PurchaseBundler.sol

    function _buy(bytes[] calldata executionData) private returns
(uint256[] memory) {
        bytes[] memory encodedOutput =
_multiSourceLoan.multicall(executionData);
...
        uint256 remainingBalance;
        assembly {
            remainingBalance := selfbalance()
        }
        if (remainingBalance != 0) {
 |>         (bool success,) = payable(msg.sender).call{value:
remainingBalance}("");
            if (!success) {
                revert CouldNotReturnEthError();
            }
        }
...
```

```
    }
```

- PurchaseBundler.sol#L446 Flows: PurchaseBundler::buy() -> _multiSourceLoan.multicall -> _multiSourceLoan.emitLoan -> PurchaseBundler::afterPrincipalTransfer -> _weth.withdraw(borrowed)

In addition, exploits are likely in a batch tx where the caller submits `emitLoan` for multiple borrowers. Suppose Borrower A has surplus principal of 0.1 ether after NFT purchase. The caller can open up a loan for self using Borrower A's 0.1 ether surplus.

Suppose a caller submits two executionData through `buy()`: (1) exeuctionData[0] for Borrower A, (2) executionData[1] for self. The surplus 0.1 ether after (1) will be used for (2).

Impacts: Borrowers can lose part of the principal, potentially to other borrowers.

Recommendation: Consider checking for surplus principal and handling refunds in `_afterPrincipalTransfer`:

```
    function _afterPrincipalTransfer(
...
        uint256 borrowed = _loan.principalAmount - _fee;
        /// @dev Get WETH from the borrower and unwrap it since listings
expect native ETH.
        _weth.withdraw(borrowed);
        (bool success, ) = executionInfo.module.call{
            value: executionInfo.value
        }(executionInfo.data);
        if (!success) {
            revert InvalidCallbackError();
        }
+      if ( borrowed > executionInfo.value) {   // accounting refunds to
_loan.borrower
...
```

Gondi: Fixed with PR-464

Zenith: Verified. Added remaining principal check and handling.

4.3 Low Risk

A total of 3 low risk findings were identified.

[L-1] Invalid whitelisted currency check

Severity: Low Status: Resolved

Target

- PurchaseBundler.sol#L149

Severity:

- Impact: Low
- Likelihood: Low

Description: Currently, a delisted currency will not revert the whitelisting check due to missing require/revert statement. The bundler is at risk of calling untrusted contracts.

```
    function approveForSwap(address currency) external {
|>          _currencyManager.isWhitelisted(currency);
        ERC20(currency).approve(address(_uniswapRouter),
type(uint256).max);
    }
```

- PurchaseBundler.sol#L149

Recommendation: Wrap in a require statement.

```
require(_currencyManager.isWhitelisted(currency), "Untrusted currency");
```

Gondi: Fixed with PR-453

Zenith: Verified. Revert condition is added.

[L-2] Multiple flows have insufficient input validations

Severity: Low Status: Resolved

Target

- [PurchaseBundler.sol#L430]
- [PurchaseBundler.sol#L185]

Severity:

- Impact: Low
- Likelihood: Low

Description: Multiple flows have insufficient input validations. (1) Missing check on executionData executionData can be empty which will bypass the intended MultiSourceLoan interaction completely and can be used to sweep funds in the contracts. For example in the buy() flow, empty executionData will bypass `emitloan` call and sweep any ETH left in the contract.

```
    function buy(bytes[] calldata executionData) external payable returns
(uint256[] memory loanIds) {
        loanIds = _buy(executionData);
    }
    function _buy(bytes[] calldata executionData) private returns
(uint256[] memory) {
|>     bytes[] memory encodedOutput =
_multiSourceLoan.multicall(executionData); //@audit-info empty
executionData will skip multicall
        uint256[] memory loanIds = new uint256[](encodedOutput.length);
        uint256 total = encodedOutput.length;
        for (uint256 i; i < total;) {
            loanIds[i] = abi.decode(encodedOutput[i], (uint256));
            unchecked {
                ++i;
            }
        }

        /// Return any remaining funds to sender.
        uint256 remainingBalance;
        assembly {
            remainingBalance := selfbalance()
        }
        if (remainingBalance != 0) {
```

```
      (bool success,) = payable(msg.sender).call{value:
remainingBalance}("");
      if (!success) {
        revert CouldNotReturnEthError();
      }
    }
    emit BNPLLoansStarted(loanIds);
    return loanIds; }
```

- PurchaseBundler.sol#L430

Note: similar in sell, executeSell, executeSellWithETH (2) Missing check on `marketPlace` executeSell doesn't check whether `marketPlace` is whitelisted before setting approval, a user might set approval to a malicious contract.

```
  function executeSell(
    ERC20[] calldata currencies,
    uint256[] calldata currencyAmounts,
    ERC721[] calldata collections,
    uint256[] calldata tokenIds,
    address marketPlace,
    bytes[] calldata executionData
  ) external {
    address buyer = _msgSender();
    for (uint256 i = 0; i < currencies.length; ++i) {
      uint256 balance = currencies[i].balanceOf(address(this));
      if (currencyAmounts[i] > balance) {
        currencies[i].safeTransferFrom(buyer, address(this),
currencyAmounts[i] - balance);
      }
|>     currencies[i].approve(marketPlace, currencyAmounts[i]);
    }
    for (uint256 i = 0; i < collections.length; ++i) {
|>      collections[i].setApprovalForAll(marketPlace, true);
    }
...
```

- PurchaseBundler.sol#L182-L185

Recommendation: Consider adding input validation to make sure executionData is not empty, and marketplace is whitelisted.

Gondi: Fixed with PR-454

Zenith: Verified.

[L-3] Incorrect order typehash

Severity: Low Status: Resolved

Target

- Hash.sol#L43-L45)

Severity:

- Impact: Low
- Likelihood: High

Description:

The `TradeMarketplace.sol` contract implements EIP712 hashing when processing marketplace orders:

```
    function _hasValidSignature(Order memory order) private view {
>>      bytes32 hash =
DOMAIN_SEPARATOR().toTypedDataHash(Hash.hash(order));
        address signer = ECDSA.recover(hash, order.signature);
        if (signer != order.maker) {
            revert InvalidSignature();
        }
    }
```

However, the `Hash.hash(order)` function uses an incorrect `encodeType`. Specifically, the `signature` field should not be included because it is not part of the encoded `hashStruct`:

```
    bytes32 private constant _TRADE_ORDER_HASH = keccak256(
        "Order(address maker,address taker,address collection,uint256
tokenId,address currency,uint256 price,uint256 nonce,uint256
expiration,bool isAsk,bytes signature)"
    );

    function hash(ITradeMarketplace.Order memory order) internal pure
returns (bytes32) {
        return keccak256(
            abi.encode(
                _TRADE_ORDER_HASH,
                order.maker,
                order.taker,
                order.collection,
                order.tokenId,
                order.currency,
                order.price,
                order.nonce,
                order.expiration,
                order.isAsk
            )
        );
    }
```

Recommendation:

```
-   bytes32 private constant _TRADE_ORDER_HASH = keccak256(
-        "Order(address maker,address taker,address collection,uint256
tokenId,address currency,uint256 price,uint256 nonce,uint256
expiration,bool isAsk,bytes signature)"
-    );
+   bytes32 private constant _TRADE_ORDER_HASH = keccak256(
+        "Order(address maker,address taker,address collection,uint256
tokenId,address currency,uint256 price,uint256 nonce,uint256
expiration,bool isAsk)"
+   );
```

Gondi: Fixed in PR-457

Zenith: Verified. The extra parameter has been removed.

4.4 Informational

A total of 4 informational findings were identified.

[I-1] Some token transfer implementation not compatible with ERC20 tokens that revert on zero transfer

Severity: Informational	Status: Resolved

Target

- PurchaseBundler.sol#L480
- PurchaseBundler.sol#L220

Severity:

- Impact: Low
- Likelihood: Low

Description: Some ERC20 tokens revert on zero value transfer. Current token transfer implementation may cause tx revert due to missing check on transfer value is not zero.

(1)

```
    function _paybackRemaining(ERC20 currency) private {
        uint256 remaining = currency.balanceOf(address(this));
|>      currency.safeTransfer(msg.sender, remaining);
    }
```

- PurchaseBundler.sol#L480 (2)

```
    function executeSellWithETH(
        bytes calldata wethPrincipalSwapData,
        ERC20 principal,
        ERC721 collection,
        uint256 tokenId,
        bytes[] calldata executionData
    ) external payable _storeMsgSender {
        _weth.deposit{value: msg.value}();
        _swapWETH(wethPrincipalSwapData);
        _sell(executionData);
        if (collection.ownerOf(tokenId) != msg.sender) {
            //// @dev we need to do this because Seaport private listings
transfer directly to recipient
            //// if the recipient is not the caller, this will fail and
revert.
            collection.safeTransferFrom(address(this), msg.sender,
tokenId);
        }
|>      principal.safeTransfer(msg.sender,
principal.balanceOf(address(this)));
    }
```

- PurchaseBundler.sol#L220

Recommendation: Consider adding a check to only transfer when value is greater than 0.

Gondi: PR-446

Zenith: Verified. Added check that transfer value is greater than 0.

[I-2] Remove unused imports

Severity: Informational Status: Resolved

Target

- PurchaseBundler.sol#L25
- PurchaseBundler.sol#L4

Severity:

- Impact: Low
- Likelihood: Low

Description: There are unused imports. (1)

```
import "forge-std/console.sol";
```

- PurchaseBundler.sol#L25 (2)

```
import "@seaport/seaport-types/src/lib/ConsiderationStructs.sol";
```

- PurchaseBundler.sol#L4

Recommendation: Remove unused imports.

Gondi: PR-450

Zenith: Verified. Unused imports were removed.

[I-3] _paybackRemainingWeth function is not used in the bundler

Severity: Informational	**Status:** Resolved

Target

- PurchaseBundler.sol#L483-L487

Severity:

- Impact: Low
- Likelihood: Low

Description: The `_paybackRemainingWeth` function is not used in the bundler contract. Additionally, it uses the deprecated `transfer` function to send ETH tokens to the buyer:

```
    function _paybackRemainingWeth() private {
        uint256 remaining = _weth.balanceOf(address(this));
        _weth.withdraw(remaining);
>>      payable(msg.sender).transfer(remaining);
    }
```

Using the `transfer` function can cause issues because a buyer's contract might have custom logic in its `receive` function. In such cases, the 2300 gas limit imposed by transfer may be insufficient to execute the transaction.

Recommendation:

- remove the function if it is not needed OR
- replace `transfer` with `call` for transferring ETH

Gondi: Fixed in PR-459

Zenith: Verified. `_paybackRemainingWeth` is now used to repay any remaining WETH in `executeSellWithEth`.

[I-4] Unnecessary code - order.taker is checked twice

Severity: Informational	Status: Resolved

Target

- [TradeMarketplace.sol#L103]
- [TradeMarketplace.sol#L38]

Severity:

- Impact: Low
- Likelihood: Low

Description: `order.taker` address is checked twice in _executeOrder. This is unnecessary.

```
    function _executeOrder(Order memory order, address taker) internal {
        _isValidOrder(order);

        if (order.taker != address(0) && order.taker != taker) {
            revert InvalidTaker();
        }
```

- TradeMarketplace.sol#L38

```
    function _isValidOrder(Order memory order) private view {
...
        if (order.taker != address(0) && order.taker != msg.sender) {
            revert InvalidTaker();
        }
```

- TradeMarketplace.sol#L103

Case1: When executeOrder is called directly, `taker` is always msg.sender. `_isValidOrder` and `_executeOrder` have duplicated checks on order.taker.

Case2: When _executeOrder is called through PurchaseBundler's flow, `taker` is address(this). `_isValidOrder` and `_executeOrder` will have conflicted order.taker check, which requires order.taker to be set as address(0).

But order.taker can also be set as address(0) in Case1, which indicates the order.maker allows any takers. I don't see a strong reason in Case2 for the double check either.

Recommendation: Consider only keeping the order.taker check in _executeOrder.

Gondi: Fixed with PR-460

Zenith: Verified. Unnecessary check is removed.