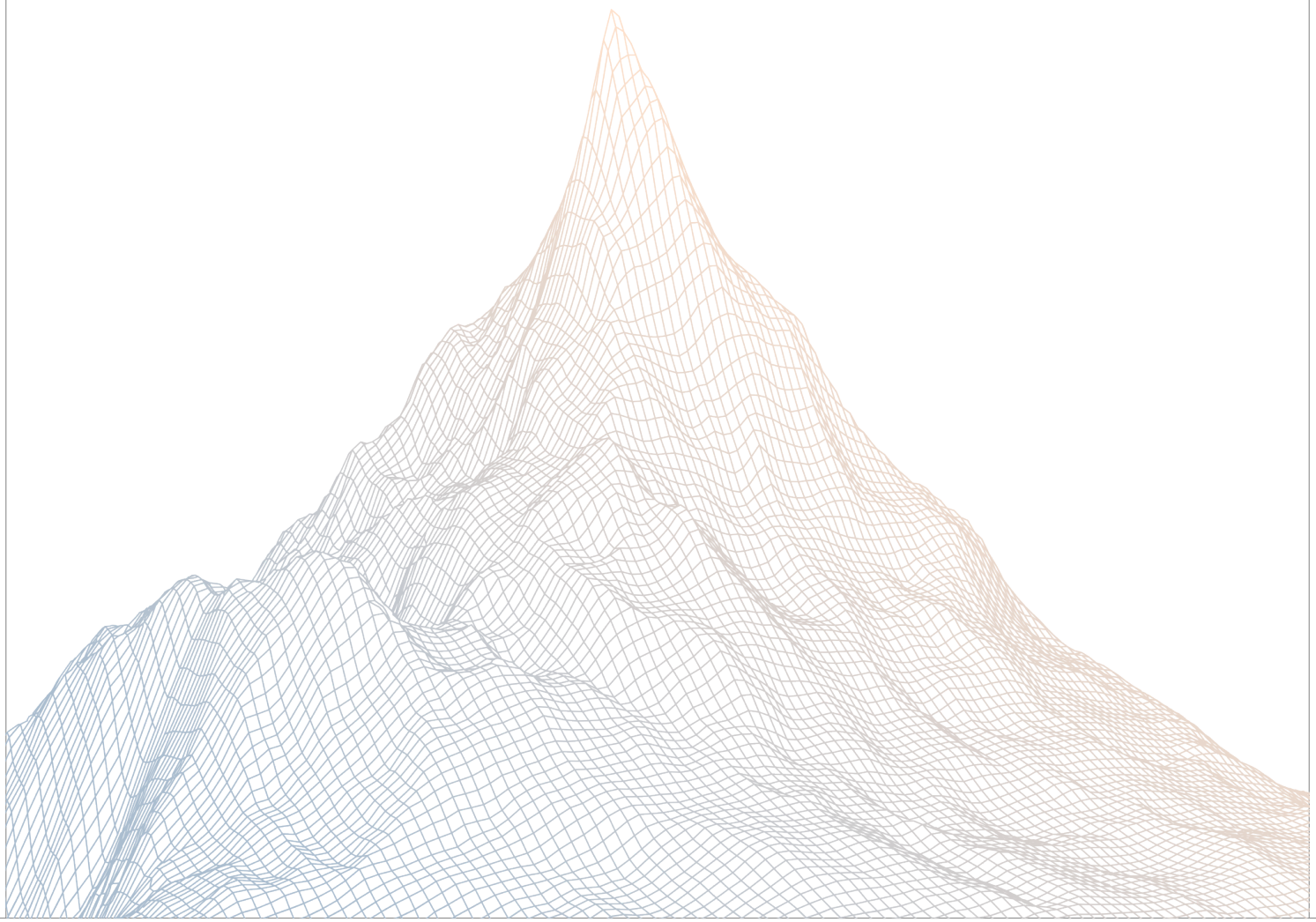
Zenith

Gondi

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: January 30th to February 3rd, 2026

AUDITED BY: oakcobalt
said

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

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 Protocol

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

GONDI V3 retains everything users love about GONDI, including pro-rata interest, instant loan refinance, and the lowest gas fees. It also introduces Tranche Seniority, enabling precise risk management. With feature upgrades and a streamlined user experience, GONDI V3 is the most advanced and efficient NFT lending protocol.

2.2 Scope

The engagement involved a review of the following targets:

Target	florida-contracts
Repository	https://github.com/pixeldaogg/florida-contracts.git
Commit Hash	`abb3858033a03b50f49ac940945cabe8fe22458b`
Files	`Diff from db695785acc5088067eea29227f2b79e09d67914`

Target	Mitigation Review
Repository	https://github.com/pixeldaogg/florida-contracts.git
Commit Hash	`67626abd8a6d65ab8992f3f84b1f1e440c1ed55c`
Files	`Diff from db695785acc5088067eea29227f2b79e09d67914`

2.3 Audit Timeline

January 30, 2026	Audit start
February 3, 2026	Audit end
February 19, 2026	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	3
Medium Risk	7
Low Risk	6
Informational	9
Total Issues	**25**

3

Findings Summary

ID	Description	Status
H-1	User-controlled purchaseAmount enables complete tax evasion	Resolved
H-2	NFT theft via _givebackNFT in executeSell	Resolved
H-3	Missing nonReentrant modifiers allow theft of proceeds or leftover funds via reentrancy	Resolved
M-1	swapAndExecute approves input currencies for target instead of output	Resolved
M-2	executeOperation and swapAndExecute ETH-native swaps fail if they are not placed first	Resolved
M-3	Removed wrappedPunk proxy support could cause certain buy flow DoS	Acknowledged
M-4	Quoter V3 passes uninitialized amountOut	Resolved
M-5	Ineffective slippage check allows bypass or DoS.	Resolved
M-6	_packRemaining can casue batch DoS or buyer fund loss due to currency mixing handling	Resolved
M-7	Missing executionInfo.value forwarding in _sellReleasedCollateralInMarketplace when contractMustBeOwner == false	Resolved
L-1	Missing currency whitelist validation in executeOperation	Resolved
L-2	finalUpdateMultiSourceLoanAddress has no timelock	Resolved
L-3	Tax updates can be applied immediately, enabling atomic exploits or transaction failure	Resolved
L-4	Insufficient validations in swapData storage	Resolved
L-5	Uncapped rawData array size can cause transaction out-of-gas	Resolved
L-6	Missing swapData size check in swapAndExecute	Resolved

ID	Description	Status
I-1	Missing early validation of Aave pool address in executeSellWithLoan	Resolved
I-2	_sell() single-item overload is dead code	Resolved
I-3	swapAndExecute uses msg.sender instead of _msgSender() for fund operations	Resolved
I-4	executeSell sets setApprovalForAll but never revokes after execution	Resolved
I-5	executeSell calls safeApprove on ETH	Resolved
I-6	Incorrect free memory pointer update in _loadSwapData causes gas inefficiency	Resolved
I-7	Empty array calls bypass validation checks	Resolved
I-8	Unused code: _swapWETH function never called	Resolved
I-9	Unnecessary code: unused rawData encoding and length caching	Resolved

4

Findings

4.1 High Risk

A total of 3 high risk findings were identified.

[H-1] User-controlled `purchaseAmount` enables complete tax evasion

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

Target

- PurchaseBundler.sol#L555
- PurchaseBundler.sol#L591

Description:

The `_handleProtocolFee` function computes the tax using `purchaseBundlerExecutionInfo.purchaseAmount`.

```
_handleProtocolFee(
    loan.borrower,
    purchaseBundlerExecutionInfo.purchaseAmount,
    address(purchaseBundlerExecutionInfo.purchaseCurrency),
    _taxes[executionInfo.module].sellTax
);
```

The value parameter comes from `purchaseBundlerExecutionInfo.purchaseAmount`, which is part of the user-supplied callback data (`_executionData`) decoded in both `_afterPrincipalTransfer` and `_afterNFTTransfer`. This callback data is embedded in the loan execution/repayment data that the borrower provides to `MultiSourceLoan`, specifically in `LoanExecutionData.executionData.callbackData` for buys and `LoanRepaymentData.data.callbackData` for sells.

```
    function emitLoan(LoanExecutionData calldata _loanExecutionData)
        external
        nonReentrant
        returns (uint256, Loan memory)
    {
```

```
        address borrower = _loanExecutionData.borrower;
        ExecutionData calldata executionData
    = _loanExecutionData.executionData;
        address principalAddress
    = _getPrincipalAddressFromExecutionData(executionData);
        address nftCollateralAddress = executionData.nftCollateralAddress;

        OfferExecution[] calldata offerExecution
    = executionData.offerExecution;

        _validateExecutionData(_loanExecutionData, borrower);
        _checkWhitelists(principalAddress, nftCollateralAddress);

        (uint256 loanId, uint256[] memory offerIds, Loan memory loan,
    uint256 totalFee) = _processOffersFromExecutionData(
            borrower,
            executionData.principalReceiver,
            principalAddress,
            nftCollateralAddress,
            executionData.tokenId,
            executionData.duration,
            offerExecution
        );

        if (_hasCallback(executionData.callbackData)) {
▷           handleAfterPrincipalTransferCallback(loan, msg.sender,
    executionData.callbackData, totalFee);
        }

        ERC721(nftCollateralAddress).transferFrom(borrower, address(this),
    executionData.tokenId);

        _loans[loanId] = loan.hash();
        emit LoanEmitted(loanId, offerIds, loan, totalFee);

        return (loanId, loan);
    }
```

Since this is entirely user-controlled, a user can set `purchaseAmount = 0` to pay zero or minimal tax, while the actual marketplace transaction can be for a much higher amount.

Recommendations:

Derive the tax base from the actual on-chain marketplace execution result rather than from user-supplied input.

Gondi: Resolved with PR-549.

Zenith: Verified.

[H-2] NFT theft via `_givebackNFT` in `executeSell`

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

Target

- PurchaseBundler.sol#L769-L770

Description:

The `executeSell` function accepts `collections` and `tokenIds` arrays that are completely decoupled from the loan repayment data in `executionData`. After executing loan repayments via `_sell(executionData)`, the function calls `_givebackNFTOrPunk` for each `collection` and `tokenId` pair.

```
function _givebackNFT(ERC721 collection, uint256 tokenId) private {
    if (collection.ownerOf(tokenId) ≠ _msgSender()) {
        collection.safeTransferFrom(collection.ownerOf(tokenId),
    _msgSender(), tokenId);
    }
}
```

This function transfers the NFT from whoever currently owns it to `_msgSender()`. Since `PurchaseBundler` is a contract that receives NFT approvals from users (via `setApprovalForAll`), an attacker can :

1. Create a minimal loan using their own NFT as collateral.
2. Call `executeSell` with their own loan repayment in `executionData`, but pass a victim's (`collection`, `tokenId`) in the `collections`/`tokenIds` arrays.
3. The `_sell` call repays the attacker's loan. Then `_givebackNFT` is called on the victim's NFT.
4. Since the victim previously approved `PurchaseBundler`, `safeTransferFrom` succeeds, transferring the victim's NFT to the attacker.

Any user who has ever granted `setApprovalForAll` to `PurchaseBundler` is at risk.

Recommendations:

Validate that each `collection` and `tokenId` pair in `executeSell` corresponds to the collateral of a loan being repaid in `executionData`.

Gondi: Resolved with PR-547.

Zenith: Verified.

[H-3] Missing nonReentrant modifiers allow theft of proceeds or leftover funds via reentrancy

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

Target

- PurchaseBundler.sol#L254-L257
- PurchaseBundler.sol#L410

Description:

`buy()` and `swapAndExeucte()` methods are missing `nonReentrant` modifiers, allowing malicious fee receivers or NFT receivers to re-enter these functions and steal proceeds or leftover swap funds from victims.

```
//@audit Missing nonreentrant modifier
function buy(
    bytes[] calldata executionData,
    address[] calldata currencies
) external payable returns (uint256[] memory loanIds) {
    loanIds = _buy(executionData);
    _paybackRemaining(currencies[0], msg.sender);
}

//@audit Missing nonreentrant modifier
function swapAndExecute(
    address[] calldata currencies,
    uint256[] calldata amountsToSwap,
    bytes calldata swapData,
    address target,
    bytes calldata executionCalldata,
    uint256 executionValue
) external payable _storeMsgSender {
...
```

NFT sale proceeds or leftover unswapped funds can be left in the contract after an NFT marketplace sell before sweeping. However, reentrancy into `buy` or `swapAndExecute` during marketplace interaction can cause such funds to be stolen by a malicious seaport fee

receiver or malicious NFT receiver. For example,

A: uswapped funds can be stolen

1. Victim calls `swapAndExecute` with currencies (e.g., 1000 USDC)
2. Swap executes: 1000 USDC → 0.5 ETH
3. Execution reaches `target.call()` - could be marketplace
4. Target pulls required funds (0.4 ETH) and sends fee to malicious fee receiver
5. Malicious fee receiver re-enters `swapAndExecute` with:
   - `currencies = [all supported currencies]`
   - `amountsToSwap = [all zeros]` (doesn't pull new funds)
   - `target = address(this)` (self-call, does nothing)
   - `executionCalldata = view function` (does nothing)
6. Re-entrant call sweeps all remaining currencies via `_paybackRemaining()`
7. Result: Victim's leftover 0.1 ETH is stolen.

The above attack path could also be performed in `buy()` in a similar manner.

B: proceeds can be stolen

1. buyer calls `executeSell`
2. `afterNFTTransfer` triggered with local market NFT sell `executeOrder` (TradeMarketPlace.sol). Buyer pays 1000 USDC to contract(purchaseBundler) and NFT is transferred to buyer contract
3. `onERC721Received` triggers buyer contract callbacks reenter `buy([], [USDC])`
   - Empty `executionData` → no loans created
   - `_paybackRemaining(USDC, buyer)` sweeps all proceeds balance
4. `afterNFTTransfer` tries to sweep proceeds currency to borrower. But all proceeds are swept by the buyer during reentrancy.
5. Tx can still succeed if borrower has enough to repay the loan. Borrower lost 1000 USDC proceeds.

Recommendations:

Consider adding `nonReentrant` modifier to both functions.

Gondi: Resolved with PR-565.

Zenith: Verified.

4.2 Medium Risk

A total of 7 medium risk findings were identified.

[M-1] `swapAndExecute` approves input `currencies` for `target` instead of output

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

Target

- PurchaseBundler.sol#L459-L465

Description:

After the swap, `swapAndExecute` approves the input `currencies` for the `target`.

```
// ...
(bool swapSuccess,) = address(_uniswapRouter).call{value:
    address(this).balance}(swapData);

// Approve input currencies for target but marketplace needs output
    currencies
if (target ≠ address(this)) {
    for (uint256 i = 0; i < currencies.length; ++i) {
        if (currencies[i] ≠ ETH) {
▷           ERC20(currencies[i]).safeApprove(target, type(uint256).max);
        }
    }
}
// ...
```

If the swap converts currency A to currency B and the marketplace needs currency B, the approval for A is useless and the marketplace call will fail due to insufficient allowance for B.

Recommendations:

Accept a separate array of currencies to approve for the marketplace target.

Gondi: Resolved with PR-556.

Zenith: Verified.

[M-2] `executeOperation` and `swapAndExecute` ETH-native swaps fail if they are not placed first

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

Target

- PurchaseBundler.sol#L382

Description:

The swap loop in `executeOperation` sends the entire ETH balance with every router call.

```
for (uint256 i = 0; i < args.swapCurrencies.length; i++) {
    // ...
    (bool swapSuccess,) = address(_uniswapRouter).call{value:
    address(this).balance}(swapData);
    // ...
}
```

If the first swap is a `ERC20` swap, all ETH is sent to the Uniswap Universal Router. When a subsequent swap needs ETH, `address(this).balance` is 0 and the swap fails.

This means multi-swap `executeSellWithLoan` calls only work if the ETH-involving swap is placed first in the array, which is an undocumented ordering constraint that will cause transaction reverts when violated.

Recommendations:

Track the ETH amount needed per swap and only forward the relevant amount.

Gondi: Resolved with PR-563.

Zenith: Verified.

[M-3] Removed wrappedPunk proxy support could cause certain buy flow DoS

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

Target

- PurchaseBundler.sol#L547-L548

Description:

Latest diff removed `_wrappedPunk.registerProxy()` from the constructor, which also removed the ability to wrap punk into `_wrappedPunk` ERC721 format.

The buy flow (`_afterPrincipalTransfer`) always wraps punks to `_c721` when purchasing from `_punkMarket`, regardless of the loan's `nftCollateralAddress`. This creates a mismatch when loans are created with `_wrappedPunk` as collateral, causing the buy flow to always fail when expected Nft collateral is wrapped punk.

```
    function _afterPrincipalTransfer(IMultiSourceLoan.Loan memory _loan,
    uint256 _fee, bytes calldata _executionData)
        private
    {
...
if (executionInfo.module == address(_punkMarket)) {
    /// @dev Wrap punk and transfer it to the borrower (loan is in
    CryptoPunks-721).
    _punkMarket.transferPunk(
        address(_punkProxy),
        _loan.nftCollateralTokenId
    );
    _c721.wrapPunk(_loan.nftCollateralTokenId);  //@audit Always wraps to
    _c721
    _c721.safeTransferFrom(
        address(this),
        _loan.borrower,
        _loan.nftCollateralTokenId
    );
}
```

In comparison, sell flow supports both _c721 and _wrappedPunk.

```
function _sellReleasedCollateralInPunkMarket(...)
    private returns (bool success) {
    ERC721 collateral = ERC721(loan.nftCollateralAddress);
    collateral.transferFrom(loan.borrower, address(this), tokenId);
    _unwrapPunk(collateral, tokenId);
    // ...
}
```

Suppose this case: buy with loan offer with `_wrappedPunk` as collateral.

1. User call buy where executionData matches a loan offer with `_wrappedPunk` as collateral(`ExecutionData.nftCollateralAddress`).
2. `afterPrincipalTransfer()` inPurchaseBundler, purchase punk from `_punkMarket` but wraps it to `_c721`.
3. In `MultiSourceLoan.emitLoan()`, nft collateral transfer would fail due to mis-matched collateral address. Tx reverts.

Recommendations:

(1) In `_afterPrincipalTransfer`, consider modifying buy flow to check `_loan.nftCollateralAddress` and wrap accordingly. (2) Add `_wrappedPunk.registerProxy()`in constructor.

Gondi: Acknowledged. We are currently making users sign a loan offer that accepts any wrapper, so the emitLoan flow would succeed with the _c721. Added a comment instead of additional validation because the contract size is getting big.

[M-4] `Quoter` V3 passes uninitialized `amountOut`

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

Target

- Quoter.sol#L17

Description:

The `quoteExactOutputV3` function receives `amountIn` as a parameter and passes `amountOut`, which is uninitialized, to `_uniswapV3Quoter.quoteExactOutput`. This means The quoter always queries for a 0-amount swap.

```
function quoteExactOutputV3(bytes memory path, uint256 amountIn)
    external returns (uint256 amountOut) {
    (amountOut,,,) = _uniswapV3Quoter.quoteExactOutput(path, amountOut);
}
```

Uniswap V3's `quoteExactOutput` signature is:

```
function quoteExactOutput(bytes memory path, uint256 amountOut)
    external returns (uint256 amountIn, ...);
```

It takes the desired output amount and returns the required input amount. Besides the uninitialized issue , the parameter is named `amountIn`, but `quoteExactOutput` expects `amountOut` (the desired output). The input parameter should represent the desired output amount, not an input amount.

This will result in `_checkSwapSlippage` always process the incorrect amount slippage.

Recommendations:

Rename the parameter, pass it correctly, and return the correct semantic value.

```
function quoteExactOutputV3(bytes memory path, uint256 amountIn) external
    returns (uint256 amountOut) {
```

```
function quoteExactOutputV3(bytes memory path, uint256 amountOut)
  external returns (uint256 amountIn) {
      (amountOut,,,) = _uniswapV3Quoter.quoteExactOutput(path, amountOut);
      (amountIn,,,) = _uniswapV3Quoter.quoteExactOutput(path, amountOut);
}
```

Gondi: Resolved with PR-558.

Zenith: Verified.

[M-5] Ineffective slippage check allows bypass or DoS.

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

Target

- PurchaseBundler.sol#L898

Description:

The `_swapAndCheck` function performs slippage validation by comparing actual swap results against a quote obtained in the same transaction. However, the implementation has multiple vulnerabilities that render the slippage check ineffective or allow bypasses:

1. Atomic quote-swap execution.

The quote and swap execute in the same transaction. If quoteData and swapData use the same route/pool, both operations see identical pool state. This makes the slippage check compare the swap result against a quote from the same state, which will always pass. This can cause unbounded slippage loss.

2. Amount not correlated - only ratio checked.

```
function _checkSwapSlippage(
    uint256 amountIn,
    uint256 amountOut,
    uint256 quotedAmountIn,
    uint256 quotedAmountOut,
    uint256 maxSlippage
) private {
    if (
        (amountOut * quotedAmountIn) * MAX_SLIPPAGE_DENOMINATOR <
        maxSlippage * (amountIn * quotedAmountOut)
    ) {
        revert TooMuchSlippageError();
    }
}
```

The formula checks (amountOut / amountIn) >= (quotedAmountOut / quotedAmountIn) * (1 - slippage), but doesn't validate that the actual amounts match the expected amounts. A buyer can swap minimal amounts (e.g., 0.01 ETH instead of 1 ETH) as long as the rate ratio matches. For example,

```
Borrower expects: 1 ETH → 3000 USDC
Buyer swaps: 0.01 ETH → 30 USDC
Result: Check passes
Borrower receives: 0.99 ETH + 30 USDC (borrower can be forced to transfer
    more USDC to cover the shortfall due to insufficient swap)
```

3. Zero swap edge case

When amountIn = 0 and amountOut = 0, the slippage check always passes. Buyer can provide swapData that does nothing (or swap fails silently), and the slippage check will pass. Borrower receives purchaseCurrency back without any swap occurring, forced to pay more out of pocket.

4. ExactOutput/ExactInput mismatch

For example, borrower quotes exactOutput (needs 1 ETH, quotes 3000 USDC input needed), but buyer swaps exactInput (swaps 3000 USDC, gets variable output 0.5 ETH). In slippage check, quote input 3000 USDC would be used as `quoteAmountOut`, and 0.5 ETH is `amountOut`. The swap is executed at a much worse price but slippage still passes, causing unbounded slippage loss.

Recommendations:

1. Consider allowing borrower to input `quoteAmountIn` and the desired minimal `quoteAmountOut` instead of using the atomic quote->swap execution
2. And consider adding an exact amount match between `quoteAmountIn` and the swap amount.
3. And check and reject when the swap amounts are zero.

Gondi: Resolved with PR-566. Design choice: If the swap is not necessary, then it's better for the borrower, as they won't have to pay fees + slippage.

Zenith: Verified, and acknowledged the design choice.

[M-6] _packRemaining can casue batch DoS or buyer fund loss due to currency mixing handling

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

Target

- PurchaseBundler.sol#L610-L611

Description:

In `_afterNFTTransfer`, `_paybackRemaining` sweeps the entire balance of a currency to a recipient without distinguishing between different fund sources. This is vulnerable when the callback is from `executeSell` with batched `executionData`.

`_paybackRemaining` and `_sendCurrency` use `_getBalance(currency)`, which returns the entire contract balance without tracking the source of funds. When a buyer/caller provided batched `bytes[] calldata executionData`, all currencies needed to purchase the NFT collaterals would be provided upfront. In _afterNFTTransfer, this causes:

1. Fund loss: Buyer's unused purchaseCurrency is stolen by borrowers

2. Batch DoS: Buyer could have provided `purchaseCurrency` for multiple loans, or a loan's `loanCurrency` could be the same as `purchaseCurrency` for a subsequent loan in the batch. This means buyer's unused purchaseCurrency can be swept entirely by the borrower of the first loan in the batch that uses the same currency as `loanCurrency` or `purchaseCurrency`. This would cause subsequent `afterNFTTransfer` callbacks for subsequent loans in the batch to fail.

```
function _afterNFTTransfer(...) private {
...
            success = _sellReleasedCollateralInMarketplace(loan,
    executionInfo, contractMustBeOwner);
        if (!success) {
            revert InvalidCallbackError();
        }
...
    if (purchaseBundlerExecutionInfo.swapData.length > 0) {
        (uint256 purchaseCurrencyBalance, uint256 loanCurrencyBalance)
```

```
    = _swapAndCheck( ... );

        ERC20(loanCurrency).safeTransfer(loan.borrower,
    loanCurrencyBalance);

▷       _sendCurrency(
          purchaseCurrency, //@audit Could include purchase currency
    supplied by buyer for subsequent loans
          loan.borrower,
          purchaseCurrencyBalance  // Includes buyer's unused!
        );
    } else {

▷       _paybackRemaining(loanCurrency, loan.borrower);  //@audit Could
    include purchase currency supplied by buyer for subsequent loans
    }
...
```

Suppose this attack scenario:

1. Buyer calls executeSell:
   - Transfers 1000 USDC to contract
   - executionData = [loan1 repayment, loan2 repayment]
2. Loan1 callback:
   - After marketplace sell, contract sends all USDC to borrower1 (includes buyer's unused)
   - Contract USDC balance is 0
3. Loan2 callback:
   - Tries to sell NFT2
   - Needs purchaseCurrency but contract has 0 USDC. The entire batch reverts.

Recommendations:

Consider parsing the marketplace return value to extract the exact proceeds of the NFT sale that was sent to the purchaseBundler contract. This would need market-specific implementation. For seaport `matchOrder`, the `returndata` would include each item(currency)'s recipient, token, and amount, which can be parsed to send the correct amount back to the borrower.

Gondi: Resolved with PR-567.

Zenith: Verified. Note there are two design constraints:

1. Buyer must send/borrow exactly what's needed, the unused portion/assets can be swept to the borrower. Consider adding documentation;
2. Tax handling: this is a separate issue that can be mitigated differently.

[M-7] Missing `executionInfo.value` forwarding in `_sellReleasedCollateralInMarketplace` when `contractMustBeOwner == false`

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

Target

- PurchaseBundler.sol#L835

Description:

The function `_sellReleasedCollateralInMarketplace` inconsistently handles `executionInfo.value` forwarding. When `contractMustBeOwner == false`, the function does not forward ETH value to the marketplace module, while the `contractMustBeOwner == true` branch correctly forwards it. This inconsistency causes legitimate marketplace sell operations to fail when ETH payment is required by borrower.

```
    function _sellReleasedCollateralInMarketplace(
        IMultiSourceLoan.Loan memory loan,
        IReservoir.ExecutionInfo memory executionInfo,
        bool contractMustBeOwner
    ) private returns (bool success) {
        if (!contractMustBeOwner) {
▷           (success,) = executionInfo.module.call(executionInfo.data);
    //@audit Missing {value: executionInfo.value}
            return success;
        }
        ERC721 collateral = ERC721(loan.nftCollateralAddress);
        uint256 tokenId = loan.nftCollateralTokenId;

        collateral.transferFrom(loan.borrower, address(this), tokenId);
        collateral.approve(executionInfo.module, tokenId);
▷       (success,) = executionInfo.module.call{value:
    executionInfo.value}(executionInfo.data); //@audit Correctly forwards
    value
    }
```

Recommendations:

Consider updating the `contractMustBeOwner == false` branch to forward `executionInfo.value` consistently with the other branch.

Gondi: Resolved with PR-569.

Zenith: Verified.

4.3 Low Risk

A total of 6 low risk findings were identified.

[L-1] Missing currency whitelist validation in executeOperation

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

Target

- PurchaseBundler.sol#L377

Description:

The `executeOperation()` processes swap currencies from `args.swapCurrencies` without validating that each currency is whitelisted in the `_currencyManager`. An illegal asset can be used to interact with a protocol contract.

```
for (uint256 i = 0; i < args.swapCurrencies.length; i++) {
    bytes memory swapData = args.swapData[i];
    address currency = address(args.swapCurrencies[i]);//@audit No check
    that the swapCurrencies are whitelisted.
    uint160 amount = args.swapAmounts[i];
    ERC20(currency).safeTransferFrom(_buyer, address(this), amount);

    _permit2.approve(currency, address(_uniswapRouter), amount, 0);
    (bool swapSuccess,) = address(_uniswapRouter).call{value:
    address(this).balance}(swapData);
    if (!swapSuccess) {
        revert InvalidCallbackError();
    }
```

Recommendations:

Consider adding currency whitelist validation in the swap loop.

Gondi: Resolved with PR-554.

Zenith: Verified.

[L-2] `finalUpdateMultiSourceLoanAddress` has no timelock

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

Target

- PurchaseBundler.sol#L628-L637

Description:

The `MultiSourceLoan` address update follows a two-step pattern but has no time delay between steps

```
function updateMultiSourceLoanAddressFirst(address _newAddress)
    external onlyOwner {
    _newAddress.checkNotZero();
    _pendingMultiSourceLoanAddress = _newAddress;
    emit MultiSourceLoanPendingUpdate(_newAddress);
}

function finalUpdateMultiSourceLoanAddress(address _newAddress)
    external onlyOwner {
    if (_pendingMultiSourceLoanAddress ≠ _newAddress) {
        revert InvalidAddressUpdateError();
    }
    _multiSourceLoan = MultiSourceLoan(_newAddress);
    _pendingMultiSourceLoanAddress = address(0);
    emit MultiSourceLoanUpdated(_newAddress);
}
```

Both functions can be called in the same transaction.

Recommendations:

Add a timelock between `updateMultiSourceLoanAddressFirst` and `finalUpdateMultiSourceLoanAddress`, similar to the other update pattern.

Gondi: Resolved with PR-562.

Zenith: Verified.

[L-3] Tax updates can be applied immediately, enabling atomic exploits or transaction failure

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

Target

- const.sol#L6

Description:

The `PURCHASE_BUNDLER_TAX_UPDATE_NOTICE` constant is set to `0` in `src/const/default/const.sol`, allowing tax updates to be applied immediately without any notice period. This enables several attack vectors:

1. Atomic tax exploit: malicious or compromised owner can set taxes to maximum (50%) and immediately apply them to extract excessive value from borrowers. And set the taxes back to normal afterwards.
2. Unexpected tax charge: Borrowers may be charged significantly higher taxes than expected, leading to financial losses
3. Transaction reverts: if tax increases settles before buy/sell tx, and borrower doesn't have enough balance to cover the tax increase portion. tx reverts.

```
uint256 constant PURCHASE_BUNDLER_TAX_UPDATE_NOTICE = 0;
```

Recommendations:

Consider changing hardcoded `PURCHASE_BUNDLER_TAX_UPDATE_NOTICE` to match other update mechanism such as 7 days.

Gondi: Resolved with PR-555.

Zenith: Verified.

[L-4] Insufficient validations in swapData storage

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

Target

- PurchaseBundler.sol#L165-L166
- PurchaseBundler.sol#L187-L188

Description:

The `_saveOneSwapData` function and `_saveSwapData` modifier lack input validations that could lead to invalid state, or potential data corruption - Empty swap data can be saved to transient storage, and there are no protections against overwriting swap data in the same transaction.

1. Empty swapData validation missing

The `_saveOneSwapData` function only validates the maximum length but does not check for empty data. When empty swap data is later loaded in `_swapAndCheck`, it causes a revert. It's better to revert illegal state writes early in the transaction.

```
function _saveOneSwapData(bytes memory data, uint256 index) private {
    uint256 length = data.length;
   //@audit missing check length is not zero
    if (length > _SWAPDATA_MAXBYTES) {
        revert InvalidSwapDataLengthError();
    }
```

2. No protection against re-write in same transaction

The `_saveSwapData` modifier does not validate whether swap data has already been written to the same indices in the current transaction. This currently could be vulnerable in a batch tx. For example, each subsequent `sell()` or `executeSell()` enters a fresh reentrancy guard states but existing swapdata can be rewritten and corrupted.

```
modifier _saveSwapData(bytes[] memory rawData) {
    //@audit No validations that swapData hasn't been written in the same tx.
    Risk of swapData re-write or corruption if attack paths allow.
    for (uint256 i = 0; i < rawData.length; i++) {
        _saveOneSwapData(rawData[i], i);
    }
    _;
}
```

Example:

1. Client contract batches two `sell()` calls in one transaction
2. First `sell()` saves swapData[0..2] at indices 0, 1, 2; execution counter becomes 3 after callbacks
3. Second `sell()` saves swapData[0..1] at indices 0, 1 (overwriting first call's data); execution counter continues from 3 which is empty even though new swapData have been saved. tx revert.

Recommendations:

Consider adding validations to reject empty swap data and prevent overwrites.

Gondi: Resolved with PR-559.

Zenith: Verified.

[L-5] Uncapped `rawData` array size can cause transaction out-of-gas

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

Target

- PurchaseBundler.sol#L187

Description:

The `_saveSwapData` modifier accepts an uncapped `bytes[] memory rawData` array. While individual swap data elements are limited to `_SWAPDATA_MAXBYTES` (8192 bytes), there is no limit on the array length itself. When a large array swap array is passed, transactions could run out of gas during the storage loop.

```
modifier _saveSwapData(bytes[] memory rawData) {
    bytes memory data = abi.encode(rawData);
    uint256 length = data.length;

    for (uint256 i = 0; i < rawData.length; i++) {
        _saveOneSwapData(rawData[i], i); //@audit  Each element is capped,
    but array size is not
    }
    _;
}
```

Recommendations:

Consider adding a maximum array size check to the `_saveSwapData` modifier.

Gondi: Resolved with PR-561.

Zenith: Verified.

[L-6] Missing swapData size check in swapAndExecute

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

Target

- PurchaseBundler.sol#L406

Description:

The `swapAndExecute` function lacks validation on the `swapData` parameter size, while other functions in the contract enforce a maximum size limit of `_SWAPDATA_MAXBYTES`. This creates an inconsistency and potential gas DoS vector.

```
function swapAndExecute(
    address[] calldata currencies,
    uint256[] calldata amountsToSwap,
    bytes calldata swapData,  //@audit No size check
    address target,
    bytes calldata executionCalldata,
    uint256 executionValue
) external payable _storeMsgSender {
    // ...
    if (swapData.length > 0) {
        // ...
        (bool swapSuccess, ) = address(_uniswapRouter).call{
            value: address(this).balance
        }(swapData);  //@audit Directly passed without size validation
    }
}
```

Recommendations:

Consider adding a max size check for `swapData` in `swapAndExecute`.

Gondi: Resolved with PR-560.

Zenith: Verified.

4.4 Informational

A total of 9 informational findings were identified.

[I-1] Missing early validation of Aave pool address in executeSellWithLoan

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

Target

- PurchaseBundler.sol#L334

Description:

The `executeSellWithLoan()` calls `flashLoan()` on `args.borrowArgs.pool` without validating that the provided pool address matches the canonical Aave V3 pool address from the `PoolAddressesProvider`. Although flashloan pool address is checked in the callback `executeOperation`. Current implementation still allows unauthorized contract calls bypassing `executeOperation` callback.

```
    function executeSellWithLoan(ExecuteSellWithLoanArgs calldata args)
    external payable _storeMsgSender {
...
        args.borrowArgs.pool
            .flashLoan(
                address(this),
                args.borrowArgs.assets,
                args.borrowArgs.amounts,
                interestRateModes,
                address(this),
                params,
                0
            );
```

Recommendations:

Consider adding pool address validation in `executeSellWithLoan()` before calling `flashLoan()`.

Gondi: Resolved with PR-550.

Zenith: Verified.

[I-2] `_sell()` single-item overload is dead code

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

Target

- PurchaseBundler.sol#L790-L800

Description:

The single-item `_sell` is never called anywhere in the codebase. Only `_sell(bytes[])` is called from `sell` and `executeSell`. The single-item overload is dead code.

Recommendations:

Remove the unused `_sell(bytes)` overload to avoid confusion.

Gondi: Resolved with PR-546.

Zenith: Verified.

[I-3] `swapAndExecute` uses `msg.sender` instead of `_msgSender()` for fund operations

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

Target

- PurchaseBundler.sol#L419

Description:

`swapAndExecute` uses the `_storeMsgSender` modifier but inconsistently uses `msg.sender` for fund operations.

```
function swapAndExecute(
    // ...
) external payable _storeMsgSender {
    // ...
▷ address caller = msg.sender;

    // Pull currencies from caller (msg.sender)
    ERC20(currency).safeTransferFrom(caller, address(this), amount
    - balance);

    // Return remaining funds to caller (msg.sender)
    _paybackRemaining(currency, caller);
    _paybackRemaining(ETH, caller);
}
```

All other entry points that use `_storeMsgSender` consistently use `_msgSender()` for similar operations.

Recommendations:

Replace `msg.sender` with `_msgSender()`.

Gondi: Resolved with PR-545.

Zenith: Verified.

[I-4] `executeSell` sets `setApprovalForAll` but never revokes after execution

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

Target

- PurchaseBundler.sol#L311

Description:

The `executeSell` function grants the `marketplace` `setApprovalForAll` for each collection but never revokes it. After execution completes, the marketplace retains approval to transfer any token from those collections owned by `PurchaseBundler`.

```
// ...
for (uint256 i = 0; i < collections.length; ++i) {
    collections[i].setApprovalForAll(marketPlace, true);
}
// ...
```

Recommendations:

Add a cleanup loop that revokes `setApprovalForAll` after execution.

Gondi: Resolved with PR-553.

Zenith: Verified.

[I-5] `executeSell` calls `safeApprove` on ETH

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

Target

- PurchaseBundler.sol#L318

Description:

When `currencies[i] == ETH` (`0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`), the function skips `safeApprove` in the first loop (`isERC20` is checked).

```
// ...
        for (uint256 i = 0; i < currencies.length; ++i) {
            address currency = currencies[i];
            bool isERC20 = currency != ETH;
            if (!_currencyManager.isWhitelisted(address(currency)) &&
    isERC20) {
                revert CurrencyNotWhitelisted();
            }
            uint256 balance = _getBalance(currency);
            bool notEnoughBalance = currencyAmounts[i] > balance;
            if (isERC20) {
                if (notEnoughBalance) {
                    ERC20(currency).safeTransferFrom(buyer, address(this),
    currencyAmounts[i] - balance);
                }
▷               ERC20(currency).safeApprove(marketPlace, currencyAmounts[i]);
            } else {
                if (notEnoughBalance) {
                    revert InvalidStateError(); // @dev we can't ask for eth,
    so it should be in the value sent
                }
            }
        }
// ...
```

but in the cleanup loop calls `safeApprove` on the ETH sentinel address unconditionally.

```
// ...
        for (uint256 i = 0; i < currencies.length; ++i) {
            _paybackRemaining(currencies[i], _msgSender());
            ERC20(currencies[i]).safeApprove(marketPlace, 0);
        }
// ...
```

solmate's `SafeTransferLib.safeApprove` does not check the target's code size. calling approve on a codeless address silently succeeds as a no-op (empty returndata is treated as success). This means `executeSell` works with ETH but wastes gas on a useless low-level call.

Recommendations:

Add an `isERC20` check in the cleanup loop, consistent with the first loop.

Gondi: Resolved with PR-564.

Zenith: Verified.

[I-6] Incorrect free memory pointer update in `_loadSwapData` causes gas inefficiency

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

Target

- PurchaseBundler.sol#L217

Description:

The `_loadSwapData` function in `PurchaseBundler.sol` incorrectly updates the free memory pointer by adding an extra 32 bytes beyond what is actually allocated. This causes unnecessary memory waste and gas inefficiency, particularly in transactions that process multiple loan repayments.

```
function _loadSwapData(uint256 index)
    private view returns (bytes memory result) {
    uint256 length;
    assembly {
        let baseSlot := add(0x40, mul(index, _SWAPDATA_MAXBYTES))
        length := tload(baseSlot)
    }
    assembly {
        result := mload(0x40) // allocate new bytes array
        mstore(result, length)
        let baseSlot := add(0x40, mul(index, _SWAPDATA_MAXBYTES))
        let ptr := add(result, 0x20)
        for {
            let i := 0
        } lt(i, length) {
            i := add(i, 0x20)
        } {
            let chunkSlot := add(baseSlot, add(1, div(i, 0x20)))
            let chunk := tload(chunkSlot)
            mstore(add(ptr, i), chunk)
        }
        //@audit  This adds extra 0x20 bytes that are never used.
        mstore(0x40, add(add(ptr, length), 0x20)) // update free mem ptr
    }
```

```
}
```

Recommendations:

Consider only rounding up to the next 32-byte boundary when update free memory pointer.

Gondi: Resolved with PR-552.

Zenith: Verified.

[I-7] Empty array calls bypass validation checks

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

Target

- PurchaseBundler.sol#L693
- PurchaseBundler.sol#L778

Description:

The `_buy` and `_sell` functions in `PurchaseBundler.sol` allow empty `executionData` arrays to bypass critical validation checks. When an empty array is passed, the validation loop that checks `executionData[i].length ⩽ 4` never executes, allowing the functions to complete successfully without performing any meaningful operations. This could be used to allow validation bypass in reentrancy attacks.

```
function _buy(bytes[] calldata executionData) private returns (uint256[]
    memory) {
    bytes[] memory encodedOutput
    = _multiSourceLoan.multicall(executionData);
    uint256[] memory loanIds = new uint256[](encodedOutput.length);
    uint256 total = encodedOutput.length;
    //@audit Empty array bypasses the validation check below
    for (uint256 i; i < total;) {
        if (executionData[i].length <= 4) {
            // it should include the selector and parameters
            revert InvalidExecutionData();
        }
```

```
function _sell(bytes[] calldata executionData) private {
    _multiSourceLoan.multicall(executionData);
    uint256[] memory loanIds = new uint256[](executionData.length);
    uint256 total = executionData.length;
    //@audit Empty array bypasses the validation check below
    for (uint256 i = 0; i < total; ++i) {
        if (executionData[i].length <= 4) {
            // it should include the selector and parameters
            revert InvalidExecutionData();
```

```
        }
```

Recommendations:

Consider adding explicit checks to reject empty arrays at the beginning of both functions.

Gondi: Resolved with PR-557.

Zenith: Verified.

[I-8] Unused code: _swapWETH function never called

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

Target

- PurchaseBundler.sol#L849-L854

Description:

The `_swapWETH` function is defined as a private function in `PurchaseBundler.sol` but is never called anywhere in the codebase, including parent contracts like `TradeMarketplace.sol`. This is dead code that should be removed to improve code maintainability and reduce contract size.

```
function _swapWETH(bytes calldata swapData) private {
    if (swapData.length == 0) return;
    _permit2.approve(
        address(_weth),
        address(_uniswapRouter),
        type(uint160).max,
        0
    );
    (bytes memory commands, bytes[] memory inputs, uint256 deadline) = abi
        .decode(swapData, (bytes, bytes[], uint256));
    _uniswapRouter.execute(commands, inputs, deadline);
}
```

Recommendations:

Consider removing the unused `_swapWETH` function entirely.

Gondi: Resolved with PR-544.

Zenith: Verified.

[I-9] Unnecessary code: unused rawData encoding and length caching

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

Target

- PurchaseBundler.sol#L184-L185

Description:

The `_saveSwapData` modifier encodes the `rawData` array and caches its byte length, but these values are never used. The loop iterates directly over the `rawData` array elements instead.

```
modifier _saveSwapData(bytes[] memory rawData) {
▷     bytes memory data = abi.encode(rawData);
▷     uint256 length = data.length;

    for (uint256 i = 0; i < rawData.length; i++) {
        _saveOneSwapData(rawData[i], i);  // Uses rawData[i] directly, not
    data
    }
    _;
}
```

Unnecessary `abi.encode()` operation consumes gas without providing any benefit

Recommendations:

Consider removing the unused encoding and length variables.

Gondi: Resolved with PR-543.

Zenith: Verified.