OMNISCIA

June 12, 2023

SMART CONTRACT AUDIT REPORT

Limit Break Marketplace


omniscia.io


info@omniscia.io


Online report: limit-break-marketplace

Marketplace Implementation Security Audit

Audit Report Revisions

Commit Hash	Date	Audit Report Hash
73d310058c	June 9th 2023	b41209d12f
0e29e681b8	June 10th 2023	f00dbcda1f
0e29e681b8	June 12th 2023	3fbe89b578

Audit Overview

We were tasked with performing an audit of the Limit Break codebase and in particular their novel marketplace module meant to faciliate royalty-enforcing EIP-721 and EIP-1155 exchanges entirely on-chain via a signature-based system.

Over the course of the audit, we identified a flaw in the batch sale processing mechanism that renders it susceptible to denial-of-service attacks, a way to bypass the exchange whitelist mechanism, as well as a potential sale order argument that can be manipulated.

We advise the Limit Break team to closely evaluate all minor-and-above findings identified in the report and promptly remediate them as well as consider all optimizational exhibits identified in the report.

As an additional note, we would like to state that the contract's code relies on certain cross-function assumptions that are not always clear (i.e. in how bundle listings and sweep collections validate their arguments).

A prime example of this is the `unchecked` accumulation of values in `PaymentProcessor::_computeAndDistributeProceedsBundle` which will never underflow due to the checked arithmetic performed in `PaymentProcessor::_validateBundledItems`.

Such cross-function security assumptions should be clearly denoted via in-line documentation where applicable to increase the code's legibility and aid future auditing efforts of the codebase.

Post-Audit Conclusion

The Limit Break team iterated through all findings within the report and provided us with a revised commit hash to evaluate all exhibits on.

We evaluated all alleviations performed by Limit Break and have identified that all exhibits identified in our original audit report have been adequately alleviated.

The Limit Break team opted to acknowledge exhibit PPD-01C as they believe the current accuracy of the variable to be more than sufficient for the blockchain implementations the contract will be deployed in.

During the remediation round, the Limit Break team procured and evaluated audit reports produced by their other audit partners. The changes performed for these exhibits were shared with us and we identified a flawed remediation action which we advised the Limit Break team to re-evaluate in relation to EIP-20 assets and mandating that they possess a `decimals` member exposed.

Additionally, a new feature was introduced to the codebase that permits payment price bounds to be specified per collection as well as specific NFT ID. We have shared teh following feedback in relation to the changes introduced to support this feature:

Post-Audit Conclusion (73d310058c)

The Limit Break team evaluated our feedback and proceeded to rectify the `decimals` related resolution they had introduced as well as assimilate our feedback in relation to the new feature.

No outstanding issues remain in the latest iteration of the codebase evaluated within the audit report and all outputs of the report have been competently consumed by the Limit Break team.

Post-Audit Conclusion (0e29e681b8)

The Limit Break team introduced a new argument to the `PaymentProcessor::constructor` that ensures ownership of the contract is transferred to that member instead of simply the creator of the contract.

As a typical static analysis output, we would advise the new argument to be sanitized as non-zero, preventing misconfigurations of the contract.

Additionally, the `Ownable::_transferOwnership` function invocation can be optimally invoked by ascertaining that `msg.sender != defaultContractOwner_` before invoking it given that such a case would result in a no-op.

Final Verdict

The Limit Break team evaluated the concerns raised in the latest post-audit conclusion chapter and has opted to acknowledge them as they plan to carefully deploy their contracts with a `defaultContractOwner_` that is expected to be different than the `msg.sender` under all cases.

As a result, we consider the audit concluded and ascertain that all exhibits outlined in the audit report have been properly consumed by the Limit Break team.

Contracts Assessed

Files in Scope	Repository	Commit(s)
Counter.s.sol (CSR)	minimum-floor-operator-omniscia	e6277f80a7, 155182e187, 73d310058c, 0e29e681b8
PaymentProcessor.sol (PPR)	minimum-floor-operator-omniscia	e6277f80a7, 155182e187, 73d310058c, 0e29e681b8
PaymentProcessorDataTypes.sol (PPD)	minimum-floor-operator-omniscia	e6277f80a7, 155182e187, 73d310058c, 0e29e681b8

Audit Synopsis

Severity	Identified	Alleviated	Partially Alleviated	Acknowledged
Unknown	2	2	0	0
Informational	7	5	1	1
Minor	5	5	0	0
Medium	3	3	0	0
Major	1	1	0	0

During the audit, we filtered and validated a total of **2 findings utilizing static analysis** tools as well as identified a total of **16 findings during the manual review** of the codebase. We strongly recommend that any minor severity or higher findings are dealt with promptly prior to the project's launch as they can introduce potential misbehaviours of the system as well as exploits.

Compilation

The project utilizes `hardhat` as its development pipeline tool, containing an array of tests and scripts coded in JavaScript.

To compile the project, the `compile` command needs to be issued via the `npx` CLI tool to `hardhat`:

```
BASH

npx hardhat compile
```

The `hardhat` tool automatically selects Solidity version `0.8.9` based on the version specified within the `hardhat.config.js` file.

The project contains discrepancies with regards to the Solidity version used, however, they are solely located in dependencies and can thus be safely ignored.

The codebase's `pragma` statements have been locked to `0.8.9` (`=0.8.9`), the same version utilized for our static analysis as well as optimizational review of the codebase.

During compilation with the `hardhat` pipeline, no errors were identified that relate to the syntax or bytecode size of the contracts.

Static Analysis

The execution of our static analysis toolkit identified **61 potential issues** within the codebase of which **54 were ruled out to be false positives** or negligible findings.

The remaining **7 issues** were validated and grouped and formalized into the **2 exhibits** that follow:

ID	Severity	Addressed	Title
PPR-01S	Informational	Partial	Illegible Numeric Value Representations
PPR-02S	Medium	Yes	Improper Invocation of EIP-20 `transferFrom`

Manual Review

A **thorough line-by-line review** was conducted on the codebase to identify potential malfunctions and vulnerabilities in Limit Break's EIP-721 / EIP-1155 marketplace mechanism.

As the project at hand implements a special-purpose marketplace meant to be used as an authorized transactor of EIP-721 / EIP-1155 assets, intricate care was put into ensuring that the **flow of funds & assets within the system conforms to the specifications and restrictions** laid forth within the protocol's specification.

We validated that **all state transitions of the system occur within sane criteria** and that all rudimentary formulas within the system execute as expected. We **pinpointed a significant denial-of-service vulnerability** within the system which could have had **moderate-to-severe ramifications** to its overall operation.

Additionally, the system was investigated for any other commonly present attack vectors such as re-entrancy attacks, mathematical truncations, logical flaws and **ERC / EIP** standard inconsistencies. The documentation of the project was satisfactory to an exemplary extent, containing extensive documentation for each function as well as data structure of the code.

One area where the documentation can be enriched is the way certain cross-function security assumptions are not immediately deducible and should be clearly denoted via in-line documentation.

A total of **16 findings** were identified over the course of the manual review of which **11 findings** concerned the behaviour and security of the system. The non-security related findings, such as optimizations, are included in the separate **Code Style** chapter.

The finding table below enumerates all these security / behavioural findings:

ID	Severity	Addressed	Title
PPR-01M	Unknown	Yes	Discrepancy of Offer Price Behaviour
PPR-02M	Unknown	Nullified	Restrictive Specification of Marketplace Fee
PPR-03M	Informational	Yes	Non-Configurational Standard Gas Limit
PPR-04M	Minor	Yes	Inexistent Validation of Prices Per Item for Buyer

ID	Severity	Addressed	Title
PPR-05M	Minor	Yes	Insufficient Distinction of Collection Level Signed Payloads
PPR-06M	Minor	Nullified	Non-Compliant EIP-2981 Deviation
PPR-07M	Minor	Yes	Non-Inclusive Evaluation of Permissions
PPR-08M	Minor	Yes	Restrictive Caller Evaluation
PPR-09M	Medium	Yes	Bypass of Marketplace Restriction
PPR-10M	Medium	Yes	Improper Boolean Flag
PPR-11M	Major	Yes	Denial-of-Service of Batch Listing Purchase

Code Style

During the manual portion of the audit, we identified **5 optimizations** that can be applied to the codebase that will decrease the operational cost associated with the execution of a particular function and generally ensure that the project complies with the latest best practices and standards in Solidity.

Additionally, this section of the audit contains any opinionated adjustments we believe the code should make to make it more legible as well as truer to its purpose.

These optimizations are enumerated below:

ID	Severity	Addressed	Title
PPR-01C	Informational	Yes	Ineffectual Usage of Safe Arithmetics
PPR-02C	Informational	Yes	Inefficient Low-Level Interactions
PPR-03C	Informational	Yes	Inefficient `mapping` Lookups
PPR-04C	Informational	Yes	Redundant Initialization of Empty Struct
PPD-01C	Informational	Acknowledged	Potential Increase of Accuracy

PaymentProcessor Static Analysis Findings

PPR-01S: Illegible Numeric Value Representations

Type	Severity	Location
Code Style	Informational	PaymentProcessor.sol:L118, L183, L195

Description:

The linked representations of numeric literals are sub-optimally represented decreasing the legibility of the codebase.

Example:

contracts/PaymentProcessor.sol

SOL

```sol
118 uint256 public constant FEE_DENOMINATOR = 10000;
```

Recommendation:

To properly illustrate each value's purpose, we advise the following guidelines to be followed. For values meant to depict fractions with a base of `1e18`, we advise fractions to be utilized directly (i.e. `1e17` becomes `0.1e18`) as they are supported. For values meant to represent a percentage base, we advise each value to utilize the underscore (`_`) separator to discern the percentage decimal (i.e. `10000` becomes `100_00`, `300` becomes `3_00` and so on). Finally, for large numeric values we simply advise the underscore character to be utilized again to represent them (i.e. `1000000` becomes `1_000_000`).

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

While the underscore separator has been introduced to the value literal, it has been done so as if representing units per thousand. In reality, the `FEE_DENOMINATOR` represents a percentage (100%) and should be represented as such (`100_00`).

PPR-02S: Improper Invocation of EIP-20 `transferFrom`

Type	Severity	Location
Standard Conformity	Medium	PaymentProcessor.sol:L1177, L1830, L1834, L1838

Description:

The linked statement does not properly validate the returned `bool` of the **EIP-20** standard `transferFrom` function. As the **standard dictates**, callers **must not** assume that `false` is never returned.

Impact:

If the code mandates that the returned `bool` is `true`, this will cause incompatibility with tokens such as USDT / Tether as no such `bool` is returned to be evaluated causing the check to fail at all times. On the other hand, if the token utilized can return a `false` value under certain conditions but the code does not validate it, the contract itself can be compromised as having received / sent funds that it never did.

Example:

contracts/PaymentProcessor.sol

SOL

```
1177paymentCoin.transferFrom(payer, payee, proceeds);
```

Recommendation:

Since not all standardized tokens are **EIP-20** compliant (such as Tether / USDT), we advise a safe wrapper library to be utilized instead such as `SafeERC20` by OpenZeppelin to opportunistically validate the returned `bool` only if it exists.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The first referenced instance properly utilizes the `SafeERC20` library to execute the transfer and the remaining three instances were replaced by an invocation of the function the newly introduced `SafeERC20::safeTransferFrom` operation is performed in.

As such, we consider this exhibit fully alleviated.

PaymentProcessor Manual Review Findings

PPR-01M: Discrepancy of Offer Price Behaviour

Type	Severity	Location
Standard Conformity	Unknown	PaymentProcessor.sol:L1324, L1459

Description:

The `offerPrice` behaviour will differ depending on which function of the `PaymentProcessor` is invoked. The **`PaymentProcessor::_executeMatchedOrderSale`** function will permit "overpayment" if the `offerPrice` exceeds the `listingMinPrice`, however, the **`PaymentProcessor::_validateBundledOffer`** function will ascertain an equality case and fail otherwise.

Impact:

This finding is not an active vulnerability and denotes a potential way the `PaymentProcessor` marketplace can be made more flexible to accommodate a wider range of use cases. As such, it is not an exploitable attack vector but rather a usability enhancement finding.

Example:

contracts/PaymentProcessor.sol

SOL

```
1459if (accumulator.sumListingPrices != bundleDetails.offerPrice) {
1460    revert PaymentProcessor__BundledOfferPriceMustEqualSumOfAllListingPrices();
1461}
```

Recommendation:

Given that the `PaymentProcessor::buyBundledListing` function could in theory be initiated by the seller (in contrast to `PaymentProcessor::sweepCollection` which is meant to be initiated by the buyer), we advise the `PaymentProcessor::_validateBundledOffer` to permit "overpayment" **solely in the** `PaymentProcessor::buyBundledListing` **case**.

This would require further adjustments in `PaymentProcessor::_computeAndDistributeProceedsBundle` which would need to disburse the full amount to the seller rather than the cumulative amount of minimum listing prices, however, we believe this to be a desirable usability enhancement for the `PaymentProcessor`.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The Limit Break team evaluated this exhibit and agreed with its conclusions, however, they have opted to retain the original logic in the codebase to avoid introducing additional complexity which in turn could result in a new defect in the system.

As the Limit Break team stated that they will keep this in mind when releasing a future version, we consider this exhibit properly acknowledged.

PPR-02M: Restrictive Specification of Marketplace Fee

Type	Severity	Location
Standard Conformity	Unknown	PaymentProcessor.sol:L1513

Description:

The marketplace fee that is imposed for each **EIP-721** / **EIP-1155** asset may be different, however, the `PaymentProcessor::_validateBundledItems` function enforces a single fee numerator per bundle purchase / collection sweep.

Impact:

This finding is not an active vulnerability and denotes a potential way the `PaymentProcessor` marketplace can be made more flexible to accommodate a wider range of use cases. As such, it is not an exploitable attack vector but rather a usability enhancement finding.

Example:

contracts/PaymentProcessor.sol

SOL

```
1501MatchedOrder memory saleDetails =
1502    MatchedOrder({
1503        sellerAcceptedOffer: false,
1504        collectionLevelOffer: false,
1505        protocol: bundleDetails.bundleBase.protocol,
1506        paymentCoin: bundleDetails.bundleBase.paymentCoin,
1507        tokenAddress: bundleDetails.bundleBase.tokenAddress,
1508        seller: seller,
1509        privateBuyer: bundleDetails.bundleBase.privateBuyer,
1510        buyer: bundleDetails.bundleBase.buyer,
1511        delegatedPurchaser: bundleDetails.bundleBase.delegatedPurchaser,
1512        marketplace: bundleDetails.bundleBase.marketplace,
1513        marketplaceFeeNumerator: bundleDetails.bundleBase.marketplaceFeeNumerator,
1514        maxRoyaltyFeeNumerator: bundledOfferItems[i].maxRoyaltyFeeNumerator,
1515        listingNonce: listingNonce,
1516        offerNonce: bundleDetails.bundleBase.offerNonce,
1517        listingMinPrice: bundledOfferItems[i].itemPrice,
1518        offerPrice: bundledOfferItems[i].itemPrice,
1519        listingExpiration: listingExpiration,
1520        offerExpiration: bundleDetails.bundleBase.offerExpiration,
1521        tokenId: bundledOfferItems[i].tokenId,
1522        amount: bundledOfferItems[i].amount
1523    });
```

Recommendation:

We advise the code to allow the specification of a `marketplaceFeeNumerator` per `bundledOfferItems` entry as an example scenario where this would be desirable would be a security policy that enforces a strict whitelist policy (i.e. not allowing even EOAs to perform sales) and contains a proprietary exchange that applies different fees per token type / token as a form of incentive.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The Limit Break team evaluated this exhibit and assessed that the gas cost it incurs does not outweigh the edge-case usability it is meant to provide. As such, we consider this exhibit nullified given that its perceived benefit was evaluated as negligible by the Limit Break team.

PPR-03M: Non-Configurational Standard Gas Limit

Type	Severity	Location
Language Specific	Informational	PaymentProcessor.sol:L183

Description:

The `PaymentProcessor::constructor` will set the default security policy's gas limit to `2_300` regardless of the blockchain the `PaymentProcessor` is deployed in.

Impact:

As a prime example, the recent Gemholic contract failure on the zkSync Era blockchain was susceptible to fund loss due to the gas stipend required for native transfers being greater than the `2_300` default one. Different blockchains have different requirements and as such, any configurational value that affects how the contract behaves at the EVM level should be adjustable during the contract's construction.

Example:

contracts/PaymentProcessor.sol

SOL

```sol
constructor(address[] memory defaultPaymentMethods) EIP712("PaymentProcessor", "1") {
    securityPolicies[DEFAULT_SECURITY_POLICY_ID] = SecurityPolicy({
        enforceExchangeWhitelist: false,
        enforcePaymentMethodWhitelist: true,
        disablePrivateListings: false,
        disableDelegatedPurchases: false,
        disableEIP1271Signatures: false,
        disableExchangeWhitelistEOABypass: false,
        pushPaymentGasLimit: 2300,
        policyOwner: address(0)
    });

    emit CreatedOrUpdatedSecurityPolicy(
        DEFAULT_SECURITY_POLICY_ID,
        false,
        true,
        false,
        false,
        false,
        false,
        2300,
        "DEFAULT SECURITY POLICY");

    for (uint256 i = 0; i < defaultPaymentMethods.length;) {
        address coin = defaultPaymentMethods[i];

        paymentMethodWhitelist[DEFAULT_SECURITY_POLICY_ID][coin] = true;
        emit CoinApprovedForPayments(DEFAULT_SECURITY_POLICY_ID, coin);

        unchecked {
            ++i;
        }
    }
}
```

Recommendation:

We advise it to be set as a configurational parameter of the `PaymentProcessor::constructor` as native transfers do not consume only `2_300` gas in all present blockchains and would cause the default security profile to fail by default.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The value of the `pushPaymentGasLimit` entry in the `SecurityPolicy` struct instantiated during the `PaymentProcessor::constructor` has been set as an argument, permitting its per-deployment configuration and thus alleviating this exhibit.

PPR-04M: Inexistent Validation of Prices Per Item for Buyer

Type	Severity	Location
Logical Fault	Minor	PaymentProcessor.sol:L1656-L1682

Description:

The `PaymentProcessor::_verifySignedOfferForBundledItems` function will not validate the accepted price-per-item and will solely validate the total amount offered as well as whether the buyer will acquire the desired `tokenIds` and `amounts`. This type of validation is weak as it permits the seller(s) to specify f.e. an abnormally high price for one `tokenId` and an abnormally low price for another, causing metrics from the `PaymentProcessor` contract to be used for price manipulation / gauging.

Impact:

When buyers specify that they wish to acquire certain assets by provisioning funds, they engage in "strategies" whereby they expect to pay a specific price per asset to properly track its upward trend and automate potential sales / purchases in their flows. By not validating the prices that the buyer accepts, it is possible to manipulate such software (including front-end implementations of marketplaces) to show abnormal increases / decreases in price and thus influence how off-chain code reacts to on-chain actions. These types of behaviours can be limited by ensuring that the price the buyer accepts per asset is validated.

Example:

contracts/PaymentProcessor.sol

SOL

```
1463_verifySignedOfferForBundledItems(
1464    keccak256(abi.encodePacked(accumulator.tokenIds)),
1465    keccak256(abi.encodePacked(accumulator.amounts)),
1466    bundleDetails,
1467    signedOffer
1468);
```

Recommendation:

We advise the `PaymentProcessor::_verifySignedOfferForBundledItems` function to also accept and validate the hash of the `accumulator.salePrices`, ensuring that the prices accepted per asset are also recognized by the buyer.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

Our recommended course of action was applied to the letter, introducing a new argument in the `PaymentProcessor::_verifySignedOfferForBundledItems` function right after the hash of the `amounts` that signifies the hash of the `salePrices`. As a result, we consider this exhibit fully alleviated.

PPR-05M: Insufficient Distinction of Collection Level Signed Payloads

Type	Severity	Location
Logical Fault	Minor	PaymentProcessor.sol:L1624

Description:

The `PaymentProcessor::_verifySignedItemOffer` function will verify the signature of the `saleDetails` payload differently depending on whether the details specify a collection-level offer or a `tokenId`, however, this mechanism is insufficient as both the **EIP-721** & **EIP-1155** standards do not impose any restriction on `tokenId` values.

Impact:

While the signed payload collision case will impact few cases, neither the **EIP-721** standard nor the **EIP-1155** standard mandate the `tokenId` values to be sequential and a `tokenId` of `type(uint256).max` is perfectly valid in such systems.

Example:

contracts/PaymentProcessor.sol

SOL

```sol
bytes32 digest =
    _hashTypedDataV4(keccak256(
        bytes.concat(
            abi.encode(
                OFFER_APPROVAL_HASH,
                saleDetails.marketplace,
                saleDetails.delegatedPurchaser,
                saleDetails.buyer,
                saleDetails.tokenAddress,
                saleDetails.collectionLevelOffer ? type(uint256).max : saleDetails.tok
                saleDetails.amount,
                saleDetails.offerPrice
            ),
            abi.encode(
                saleDetails.offerExpiration,
                saleDetails.offerNonce,
                _checkAndInvalidateNonce(
                    saleDetails.marketplace,
                    saleDetails.buyer,
                    saleDetails.offerNonce,
                    false
                ),
                saleDetails.paymentCoin
            )
        )
    )
);
```

Recommendation:

We advise the encoded payload to make use of the `collectionLevelOffer` boolean variable directly as presently, a signed offer for a `tokenId` of `type(uint256).max` and a collection-level signed offer are indistinguishable.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The Limit Break team evaluated this exhibit and opted to introduce a new `PaymentProcessor::_verifySignedCollectionOffer` function that generates the payload to be validated using a different prefix (`COLLECTION_OFFER_APPROVAL_HASH`).

The solution introduced by Limit Break achieves the same end-result as our proposed course of action, with signed collection offers and signed item offers being clearly distinguishable between them thus addressing this exhibit's concerns.

PPR-06M: Non-Compliant EIP-2981 Deviation

Type	Severity	Location
Standard Conformity	Minor	PaymentProcessor.sol:L1960-L1962

Description:

The **EIP-2981** standard does not prohibit the `royaltyReceiver` from being the zero-address. Additionally, a use-case whereby asset-level royalties are paid to the zero-address may be desirable as f.e. a proprietary token may be enforced for sales (via the `paymentMethodWhitelist` configuration) and would be expected to be burned on each sale to simulate a deflationary price action.

Impact:

As desirable use-cases exist whereby royalties would be paid to the zero-address, it is ill-advised to go against the **EIP-2981** standard as the referenced deviation provides little-to-no benefit.

Example:

contracts/PaymentProcessor.sol

SOL

```
1958(address royaltyReceiver, uint256 royaltyAmount) = abi.decode(result, (address, uint25
1959
1960if (royaltyReceiver == address(0)) {
1961    royaltyAmount = 0;
1962}
```

Recommendation:

We advise the code to omit the referenced code statement, ensuring the contract is compliant with the **EIP-2981** standard in full and permitting novel ways of integrating the Limit Break marketplace.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

After extensive deliberation between Limit Break's and Omniscia's teams, we have concluded that this particular finding is incorrect in that the deviation specified does not go against the **EIP-2981** standard.

While there may be use cases that treat the `royaltyReceiver` being zero as "valid", the Limit Break team has opted to retain their current behaviour in the codebase as they consider it a seldom case.

Given that the code of Limit Break was and remains compliant with the **EIP-2981** standard, we consider this exhibit nullified.

PPR-07M: Non-Inclusive Evaluation of Permissions

Type	Severity	Location
Logical Fault	Minor	PaymentProcessor.sol:L1203-L1210

Description:

The `PaymentProcessor::_requireCallerIsNFTOrContractOwnerOrAdmin` function is expected to evaluate whether the `msg.sender` has the necessary permissions to set the security policy for the specified `tokenAddress`, however, the permission checks are performed in sequence without necessarily evaluating all steps.

Impact:

While non-standard, certain implementations will be forced to implement both standards as part of inter-project dependencies as a dependency may implement the `Ownable` interface which cannot be removed when importing it externally.

Example:

contracts/PaymentProcessor.sol

SOL

```sol
function _requireCallerIsNFTOrContractOwnerOrAdmin(address tokenAddress) internal view
    bool callerHasPermissions = false;
    if(tokenAddress.code.length > 0) {
        callerHasPermissions = _msgSender() == tokenAddress;
        if(!callerHasPermissions) {
            try IOwnable(tokenAddress).owner() returns (address contractOwner) {
                callerHasPermissions = _msgSender() == contractOwner;
            } catch {
                try IAccessControl(tokenAddress).hasRole(DEFAULT_ACCESS_CONTROL_ADMIN_
                    returns (bool callerIsContractAdmin) {
                    callerHasPermissions = callerIsContractAdmin;
                } catch {}
            }
        }
    }

    if(!callerHasPermissions) {
        revert PaymentProcessor__CallerMustHaveElevatedPermissionsForSpecifiedNFT();
    }
}
```

Recommendation:

We advise the access control `try-catch` clause to be relocated outside the uppermost ownership `try-catch` clause, evaluating it within a new `if` clause that evaluates whether `callerHasPermissions` has not been set yet. This will permit contracts compliant with both `IOwnable` and `IAccessControl` to allow both the `owner` and the `DEFAULT_ACCESS_CONTROL_ADMIN_ROLE` to set security policies for the `tokenAddress`.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The Limit Break team evaluated this exhibit and applied its recommendation, evaluating all authorization states sequentially as long as the `Context::_msgSender` has not been authorized yet. As such, we consider this exhibit fully alleviated.

PPR-08M: Restrictive Caller Evaluation

Type	Severity	Location
Language Specific	Minor	PaymentProcessor.sol:L1200

Description:

The `PaymentProcessor::_requireCallerIsNFTOrContractOwnerOrAdmin` function evaluates that the `tokenAddress` has code specified within it when it is validated. When a contract is initially deployed in a blockchain, its `code.length` will evaluate to `0` as it has not been "constructed" yet. As such, the `PaymentProcessor` contract disallows setting the security policy of an NFT during its `ERC721::constructor` which is an ill-advised trait.

Impact:

As the `PaymentProcessor` is expected to be a standard other projects build on, it is imperative that it can be integrated with via a plethora of ways. Setting the security policy of an NFT during its `ERC721::constructor` guarantees that it is immutable, a sought-after characteristic for smart contracts that is currently impossible due to the restrictive check of `PaymentProcessor::_requireCallerIsNFTOrContractOwnerOrAdmin`.

Example:

contracts/PaymentProcessor.sol

SOL

```sol
function _requireCallerIsNFTOrContractOwnerOrAdmin(address tokenAddress) internal view
    bool callerHasPermissions = false;
    if(tokenAddress.code.length > 0) {
        callerHasPermissions = _msgSender() == tokenAddress;
        if(!callerHasPermissions) {
            try IOwnable(tokenAddress).owner() returns (address contractOwner) {
                callerHasPermissions = _msgSender() == contractOwner;
            } catch {
                try IAccessControl(tokenAddress).hasRole(DEFAULT_ACCESS_CONTROL_ADMIN_
                    returns (bool callerIsContractAdmin) {
                    callerHasPermissions = callerIsContractAdmin;
                } catch {}
            }
        }
    }

    if(!callerHasPermissions) {
        revert PaymentProcessor__CallerMustHaveElevatedPermissionsForSpecifiedNFT();
    }
}
```

Recommendation:

We advise the code to not evaluate the `code.length` at all and simply validate that the `_msgSender` is equal to the `tokenAddress`. While this will cause EOAs to also be able to set a security policy, this is inconsequential and the benefit of being able to set a security policy during a contract's `constructor` outweighs the drawback of permitting non-contract `tokenSecurityPolicies` policies.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The **`PaymentProcessor::_requireCallerIsNFTOrContractOwnerOrAdmin`** function was updated to no longer evaluate whether code is present at the `tokenAddress` and to directly evaluate whether the `Context::_msgSender` is the `tokenAddress`, alleviating this exhibit in full.

PPR-09M: Bypass of Marketplace Restriction

Type	Severity	Location
Logical Fault	Medium	PaymentProcessor.sol:L1363-L1367, L1448-L1452

Description:

The `exchangeWhitelist` whitelist can be bypassed by specialized marketplaces which perform `PaymentProcessor` transactions in the `constructor` of newly deployed contracts.

Impact:

The overhead of performing each `PaymentProcessor` operation via a freshly deployed contract is minimal if optimal patterns are utilized (i.e. `delegatecall` instructions) and as such, the `exchangeWhitelist` can be bypassed realistically.

The following contract can be freshly deployed to execute a sale bypassing an enforced exchange whitelist:

SOL

```
pragma solidity 0.8.9;

import "contracts/IPaymentProcessor.sol";

contract ExchangeWhitelistBypass {
    // Any code we execute in the constructor will cause a `code` evaluation of the `Excha
    constructor(
        IPaymentProcessor paymentProcessor,
        MatchedOrder memory saleDetails,
        SignatureECDSA memory signedListing,
        SignatureECDSA memory signedOffer
    ) {
        // Below operation will succeed regardless of whether a whitelist policy has been
        paymentProcessor.buySingleListing(
            saleDetails,
            signedListing,
            signedOffer
        );
    }
}
```

Example:

contracts/PaymentProcessor.sol

```sol
if (securityPolicy.enforceExchangeWhitelist) {
    if (_msgSender().code.length > 0) {
        if (!exchangeWhitelist[securityPolicyId][_msgSender()]) {
            revert PaymentProcessor__CallerIsNotWhitelistedMarketplace();
        }
    } else if (securityPolicy.disableExchangeWhitelistEOABypass) {
        if (!exchangeWhitelist[securityPolicyId][_msgSender()]) {
            revert PaymentProcessor__TokenSecurityPolicyDoesNotAllowEOACallers();
        }
    }
}
```

Recommendation:

Given that the exchange whitelist can be bypassed, we advise adequate warnings to be specified in the documentation of the `PaymentProcessor` contract. Alternatively, we advise the `enforceExchangeWhitelist` to simply validate all callers and to omit the `disableExchangeWhitelistEOABypass` functionality, ensuring that when an exchange whitelist is enforced all types of callers are validated regardless of whether they contain code or not.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The Limit Break team opted to instead apply a `tx.origin` equality check with the call's `Context::_msgSender`.

While this security measure is adequate in the current blockchain state, it may be invalidated in the future by EIPs such as **EIP-3074**. As such, we advise the situation to be monitored and the code's logic to be appropriately updated if the `tx.origin` check becomes insecure.

For the intents and purposes of the contract in the current context of the audit report as of the 8th of June 2023, we consider the current code secure and the exhibit alleviated in full.

PPR-10M: Improper Boolean Flag

Type	Severity	Location
Logical Fault	Medium	PaymentProcessor.sol:L1291-L1293

Description:

The `saleDetails.sellerAcceptedOffer` offer boolean flag is not documented properly within the codebase and denotes whether the seller likely initiated the transaction. In the single instance it is utilized, it prevents execution of a sale if the payment method has been specified as "native", however, this is not adequately documented and a single `PaymentProcessor__CollectionLevelOrItemLevelOffersCanOnlyBeMadeUsingERC20PaymentMethods` error is yielded.

As the boolean flag is not validated in the signature methods **`PaymentProcessor::_verifySignedItemListing`** and **`PaymentProcessor::_verifySignedItemOffer`**, a transaction that ultimately executes **`PaymentProcessor::_executeMatchedOrderSale`** can be replaced by another with the `saleDetails.sellerAcceptedOffer` set to `false` without failing. As such, there is no actual protection enforced by the variable.

Impact:

The `saleDetails.sellerAcceptedOffer` variable is entirely malleable via a transaction replay attack as it is not contained in the `SignatureECDSA` payloads, rendering the security guarantee it is meant to achieve incorrect.

This function can be included to the `ErrorCases.t.sol` file to demonstrate the vulnerability:

SOL

```
function test_demonstrateMalleabilityOfSellerAcceptedOffer() public {
        MatchedOrder memory saleDetails = MatchedOrder({
            sellerAcceptedOffer\

### Example:

```sol title=contracts/PaymentProcessor.sol highlight={15,16,17} lineNumbers=true lineOffs
function _executeMatchedOrderSale(
    uint256 msgValue,
    MatchedOrder memory saleDetails,
    SignatureECDSA memory signedListing,
    SignatureECDSA memory signedOffer
) private {
    uint256 securityPolicyId = tokenSecurityPolicies[saleDetails.tokenAddress];
    SecurityPolicy memory securityPolicy = securityPolicies[securityPolicyId];

    if (saleDetails.paymentCoin == address(0)) {
        if (saleDetails.offerPrice != msgValue) {
            revert PaymentProcessor__OfferPriceMustEqualSalePrice();
        }

        if (saleDetails.sellerAcceptedOffer) {
            revert PaymentProcessor__CollectionLevelOrItemLevelOffersCanOnlyBeMadeUsingERC
        }
    } else {
        if (msgValue > 0) {
            revert PaymentProcessor__CannotIncludeNativeFundsWhenPaymentMethodIsAnERC20Coi
        }

        _verifyPaymentCoinIsApproved(
            securityPolicyId,
            securityPolicy.enforcePaymentMethodWhitelist,
            saleDetails.tokenAddress,
            saleDetails.paymentCoin);
    }
```
```

Recommendation:

First, we advise the purpose of the flag to be clearly denoted in the codebase and documented. Afterwards, we advise the Limit Break team to revisit how the variable is utilized and integrate it within the `PaymentProcessor::_verifySignedItemListing` method to ensure that it cannot be manipulated by another user submitting the same transaction signed payloads.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The `sellerAcceptedOffer` boolean flag has become part of the `SALE_APPROVAL_HASH` payload, ensuring that it is properly included in the seller's signature and preventing the on-chain race condition of the exhibit from manifesting.

PPR-11M: Denial-of-Service of Batch Listing Purchase

Type	Severity	Location
Logical Fault	Major	PaymentProcessor.sol:L752

Description:

The `PaymentProcessor::buyBatchOfListings` function attempts to purchase a set of listings in sequence, however, in doing so it opens itself up to a denial-of-service attack whereby a malicious user detects a `PaymentProcessor::buyBatchOfListings` call and submits a single one via the `PaymentProcessor::buySingleListing` function, invalidating the nonce and thus causing the original `PaymentProcessor::buyBatchOfListings` function execution to fail.

Impact:

The operational integrity of a `PaymentProcessor::buyBatchOfListings` function is imperative to the correct operation of the Limit Break marketplace as certain **EIP-1155** / **EIP-721** assets are value-less without additional assets to support them from the same ecosystem.

The vulnerability showcased permits any user to identify a `PaymentProcessor::buyBatchOfListings` transaction and cause it to deliberately fail if it contains more than one items in its batch, significantly affecting how marketplaces integrate with the `PaymentProcessor` as well as the integrity of marketplace transactions by buyers and sellers alike.

While the `PaymentProcessor::sweepCollection` function is susceptible to the same type of race-condition, the failure of the transaction in that case is desirable as the buyer wishes to sweep the full collection and is incentivized to specify a high fee to ensure that all items within the collection are available for purchase.

This function can be included to the `ErrorCases.t.sol` file to demonstrate the vulnerability:

SOL

```
function test_demonstrateMatchedOrderBundleRaceCondition() public {
        uint256 numItemsInBundle = 10;

        MatchedOrderBundleBase
            memory bundledOfferDetails = MatchedOrderBundleBase({
                protocol\
```

Example:

```sol title=contracts/PaymentProcessor.sol highlight={42} lineNumbers=true lineOffset=710
function buyBatchOfListings(
    MatchedOrder[] calldata saleDetailsArray,
    SignatureECDSA[] calldata signedListings,
    SignatureECDSA[] calldata signedOffers
) external payable override {
    _requireNotPaused();

    if (saleDetailsArray.length != signedListings.length ||
        saleDetailsArray.length != signedOffers.length) {
        revert PaymentProcessor__InputArrayLengthMismatch();
    }

    if (saleDetailsArray.length == 0) {
        revert PaymentProcessor__InputArrayLengthCannotBeZero();
    }

    uint256 runningBalanceNativeProceeds = msg.value;

    MatchedOrder memory saleDetails;
    SignatureECDSA memory signedListing;
    SignatureECDSA memory signedOffer;
    uint256 msgValue;

    for (uint256 i = 0; i < saleDetailsArray.length;) {
        saleDetails = saleDetailsArray[i];
        signedListing = signedListings[i];
        signedOffer = signedOffers[i];
        msgValue = 0;

        if(saleDetails.paymentCoin == address(0)) {
            msgValue = saleDetails.offerPrice;

            if (runningBalanceNativeProceeds < msgValue) {
                revert PaymentProcessor__RanOutOfNativeFunds();
            }
```

```
            unchecked {
                runningBalanceNativeProceeds -= msgValue;
            }
        }

        _executeMatchedOrderSale(msgValue, saleDetails, signedListing, signedOffer);

        unchecked {
            ++i;
        }
    }

    if (runningBalanceNativeProceeds > 0) {
        revert PaymentProcessor__OverpaidNativeFunds();
    }
}
```

Recommendation:

Given that sellers may wish to sell items only as part of a bundle and buyers may wish to purchase items only as part of a bundle, the signed `SignatureECDSA` payloads should include a new flag that indicates whether the transaction being validated should be part of a batch transaction or an atomic transaction. To note, the validation should occur solely when a batch transaction is submitted (i.e. a signed payload of an atomic transaction can be part of a batch but a signed payload of a batch transaction cannot be an atomic transaction).

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The Limit Break team evaluated this exhibit and has opted for an always-partial-fill approach in their marketplace. This mechanism has only been introduced when an **EIP-20** payment method is specified as the Limit Break team did not want to handle native currency refunds in case a batch operation failed whilst the full amount for the batch had been sent in the form of native funds.

Given that batch purchase transactions using **EIP-20** assets can no longer be hijacked and the codebase has been documented extensively to highlight that a batch operation does not guarantee all items in the batch will be fulfilled, we consider this exhibit fully alleviated.

PaymentProcessor Code Style Findings

PPR-01C: Ineffectual Usage of Safe Arithmetics

Type	Severity	Location
Language Specific	Informational	PaymentProcessor.sol:L271

Description:

The linked mathematical operations are guaranteed to be performed safely by surrounding conditionals evaluated in either `require` checks or `if-else` constructs.

Example:

contracts/PaymentProcessor.sol

SOL

```
271 uint256 securityPolicyId = ++lastSecurityPolicyId;
```

Recommendation:

Given that safe arithmetics are toggled on by default in `pragma` versions of `0.8.X`, we advise the linked statements to be wrapped in `unchecked` code blocks thereby optimizing their execution cost.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The referenced increment statement has been wrapped in an `unchecked` block safely, optimizing its gas cost.

PPR-02C: Inefficient Low-Level Interactions

Type	Severity	Location
Code Style	Informational	PaymentProcessor.sol:L1954-L1955, L2005-L2008

Description:

The referenced operations represent low-level `staticcall` instructions performed for known function signatures during compilation time.

Example:

contracts/PaymentProcessor.sol

SOL

```
1954(bool success, bytes memory result) =
1955    tokenAddress.staticcall(abi.encodeWithSelector(IERC2981.royaltyInfo.selector, toke
```

Recommendation:

We advise `interface` declarations to be utilized instead in a `try-catch` construct, optimizing the legibility of the codebase significantly. As an additional note, the `interface` functions that are invoked on each respective instance **must be declared as `view`** to adequately instruct the compiler to perform these calls as `staticcall` instructions instead of `call` instructions. For more information, consult the **relevant documentation section of the Solidity language**.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The code was updated to utilize the `try-catch` syntax, greatly increasing its legibility.

PPR-03C: Inefficient `mapping` Lookups

Type	Severity	Location
Gas Optimization	Informational	PaymentProcessor.sol:L407, L411, L431, L435, L455, L459, L479, L483

Description:

The linked statements perform key-based lookup operations on `mapping` declarations from storage multiple times for the same key redundantly.

Example:

contracts/PaymentProcessor.sol

SOL

```sol
404 function whitelistExchange(uint256 securityPolicyId, address account) external overrid
405     _requireCallerOwnsSecurityPolicy(securityPolicyId);
406
407     if (exchangeWhitelist[securityPolicyId][account]) {
408         revert PaymentProcessor__ExchangeIsWhitelisted();
409     }
410
411     exchangeWhitelist[securityPolicyId][account] = true;
412     emit ExchangeAddedToWhitelist(securityPolicyId, account);
413 }
```

Recommendation:

As the lookups internally perform an expensive `keccak256` operation, we advise the lookups to be cached wherever possible to a single local declaration that either holds the value of the `mapping` in case of primitive types or holds a `storage` pointer to the `struct` contained.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

All referenced `mapping` lookups have been optimized by caching their interim lookup to a local `storage` variable that acts as a pointer, greatly reducing their gas cost.

PPR-04C: Redundant Initialization of Empty Struct

Type	Severity	Location
Code Style	Informational	PaymentProcessor.sol:L1945-L1950

Description:

The `PaymentProcessor::_computePaymentSplits` function defines a `SplitProceeds memory` return variable and declares it as empty at the beginning of the function.

Example:

contracts/PaymentProcessor.sol

SOL

```sol
1938function _computePaymentSplits(
1939    uint256 salePrice,
1940    address tokenAddress,
1941    uint256 tokenId,
1942    address marketplaceFeeRecipient,
1943    uint256 marketplaceFeeNumerator,
1944    uint256 maxRoyaltyFeeNumerator) private view returns (SplitProceeds memory) {
1945    SplitProceeds memory proceeds = SplitProceeds({
1946        royaltyRecipient: address(0),
1947        royaltyProceeds: 0,
1948        marketplaceProceeds: 0,
1949        sellerProceeds: 0
1950    });
```

Recommendation:

We advise the `returns` argument to be explicitly named (i.e. `SplitProceeds memory proceeds`) and the empty-struct declaration within the `PaymentProcessor::_computePaymentSplits` function to be omitted, utilizing the `returns` variable directly.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The redundant empty struct assignment has been safely omitted from the codebase.

PaymentProcessorDataTypes Code Style Findings

PPD-01C: Potential Increase of Accuracy

Type	Severity	Location
Code Style	Informational	PaymentProcessorDataTypes.sol:L185

Description:

The `pushPaymentGasLimit` member of the `SecurityPolicy` struct can be increased in accuracy without affecting the data's tight-packing mechanism as the current members of it sum to `240` bits.

Example:

contracts/PaymentProcessorDataTypes.sol

SOL

```sol
178 struct SecurityPolicy {
179     bool enforceExchangeWhitelist;
180     bool enforcePaymentMethodWhitelist;
181     bool disablePrivateListings;
182     bool disableDelegatedPurchases;
183     bool disableEIP1271Signatures;
184     bool disableExchangeWhitelistEOABypass;
185     uint32 pushPaymentGasLimit;
186     address policyOwner;
187 }
```

Recommendation:

We advise it to be increased by `16` bits in accuracy, upgrading its type to `uint48`.

Alleviation (155182e187db7c9614d1f3b81898a57872020ed7):

The Limit Break team evaluated this exhibit but opted to retain the current accuracy in place as greater accuracy would not be realistically utilized in a production environment.

Finding Types

A description of each finding type included in the report can be found below and is linked by each respective finding. A full list of finding types Omniscia has defined will be viewable at the central audit methodology we will publish soon.

External Call Validation

Many contracts that interact with DeFi contain a set of complex external call executions that need to happen in a particular sequence and whose execution is usually taken for granted whereby it is not always the case. External calls should always be validated, either in the form of `require` checks imposed at the contract-level or via more intricate mechanisms such as invoking an external getter-variable and ensuring that it has been properly updated.

Input Sanitization

As there are no inherent guarantees to the inputs a function accepts, a set of guards should always be in place to sanitize the values passed in to a particular function.

Indeterminate Code

These types of issues arise when a linked code segment may not behave as expected, either due to mistyped code, convoluted `if` blocks, overlapping functions / variable names and other ambiguous statements.

Language Specific

Language specific issues arise from certain peculiarities that the Solidity language boasts that discerns it from other conventional programming languages. For example, the EVM is a 256-bit machine meaning that operations on less-than-256-bit types are more costly for the EVM in terms of gas costs, meaning that loops utilizing a `uint8` variable because their limit will never exceed the 8-bit range actually cost more than redundantly using a `uint256` variable.

Code Style

An official Solidity style guide exists that is constantly under development and is adjusted on each new Solidity release, designating how the overall look and feel of a codebase should be. In these types of findings, we identify whether a project conforms to a particular naming convention and whether that convention is consistent within the codebase and legible. In case of inconsistencies, we point them out under this category. Additionally, variable shadowing falls under this category as well which is identified when a

local-level variable contains the same name as a contract-level variable that is present in the inheritance chain of the local execution level's context.

Gas Optimization

Gas optimization findings relate to ways the codebase can be optimized to reduce the gas cost involved with interacting with it to various degrees. These types of findings are completely optional and are pointed out for the benefit of the project's developers.

Standard Conformity

These types of findings relate to incompatibility between a particular standard's implementation and the project's implementation, oftentimes causing significant issues in the usability of the contracts.

Mathematical Operations

In Solidity, math generally behaves differently than other programming languages due to the constraints of the EVM. A prime example of this difference is the truncation of values during a division which in turn leads to loss of precision and can cause systems to behave incorrectly when dealing with percentages and proportion calculations.

Logical Fault

This category is a bit broad and is meant to cover implementations that contain flaws in the way they are implemented, either due to unimplemented functionality, unaccounted-for edge cases or similar extraordinary scenarios.

Centralization Concern

This category covers all findings that relate to a significant degree of centralization present in the project and as such the potential of a Single-Point-of-Failure (SPoF) for the project that we urge them to re-consider and potentially omit.

Reentrant Call

This category relates to findings that arise from re-entrant external calls (such as EIP-721 minting operations) and revolve around the inapplicacy of the Checks-Effects-Interactions (CEI) pattern, a pattern that dictates checks (`require` statements etc.) should occur before effects (local storage updates) and interactions (external calls) should be performed last.

Disclaimer

The following disclaimer applies to all versions of the audit report produced (preliminary / public / private) and is in effect for all past, current, and future audit reports that are produced and hosted under Omniscia:

IMPORTANT TERMS & CONDITIONS REGARDING OUR SECURITY AUDITS/REVIEWS/REPORTS AND ALL PUBLIC/PRIVATE CONTENT/DELIVERABLES

Omniscia ("Omniscia") has conducted an independent security review to verify the integrity of and highlight any vulnerabilities, bugs or errors, intentional or unintentional, that may be present in the codebase that were provided for the scope of this Engagement.

Blockchain technology and the cryptographic assets it supports are nascent technologies. This makes them extremely volatile assets. Any assessment report obtained on such volatile and nascent assets may include unpredictable results which may lead to positive or negative outcomes.

In some cases, services provided may be reliant on a variety of third parties. This security review does not constitute endorsement, agreement or acceptance for the Project and technology that was reviewed. Users relying on this security review should not consider this as having any merit for financial advice or technological due diligence in any shape, form or nature.

The veracity and accuracy of the findings presented in this report relate solely to the proficiency, competence, aptitude and discretion of our auditors. Omniscia and its employees make no guarantees, nor assurance that the contracts are free of exploits, bugs, vulnerabilities, deprecation of technologies or any system / economical / mathematical malfunction.

This audit report shall not be printed, saved, disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Omniscia.

All the information/opinions/suggestions provided in this report does not constitute financial or investment advice, nor should it be used to signal that any person reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report.

Information in this report is provided 'as is'. Omniscia is under no covenant to the completeness, accuracy or solidity of the contracts reviewed. Omniscia's goal is to help reduce the attack vectors/surface and the high level of variance associated with utilizing new and consistently changing technologies.

Omniscia in no way claims any guarantee, warranty or assurance of security or functionality of the technology that was in scope for this security review.

In no event will Omniscia, its partners, employees, agents or any parties related to the design/creation of this security review be ever liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this security review.

Cryptocurrencies and all other technologies directly or indirectly related to cryptocurrencies are not standardized, highly prone to malfunction and extremely speculative by nature. No due diligence and/or safeguards may be insufficient and users should exercise maximum caution when participating and/or investing in this nascent industry.

The preparation of this security review has made all reasonable attempts to provide clear and actionable recommendations to the Project team (the “client”) with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts in scope for this engagement.

It is the sole responsibility of the Project team to provide adequate levels of test and perform the necessary checks to ensure that the contracts are functioning as intended, and more specifically to ensure that the functions contained within the contracts in scope have the desired intended effects, functionalities and outcomes, as documented by the Project team.

All services, the security reports, discussions, work product, attack vectors description or any other materials, products or results of this security review engagement is provided "as is" and "as available" and with all faults, uncertainty and defects without warranty or guarantee of any kind.

Omniscia will assume no liability or responsibility for delays, errors, mistakes, or any inaccuracies of content, suggestions, materials or for any loss, delay, damage of any kind which arose as a result of this engagement/security review.

Omniscia will assume no liability or responsibility for any personal injury, property damage, of any kind whatsoever that resulted in this engagement and the customer having access to or use of the products, engineers, services, security report, or any other other materials.

For avoidance of doubt, this report, its content, access, and/or usage thereof, including any associated services or materials, shall not be considered or relied upon as any form of financial, investment, tax, legal, regulatory, or any other type of advice.