OMNISCIA

July 9, 2022

SMART CONTRACT AUDIT REPORT

Redacted Cartel BTRFLY V2


omniscia.io


info@omniscia.io


Online report: redacted-cartel-butterfly-token-v2/

Butterfly Token V2 Security Audit

Audit Overview

We were tasked with performing an audit of the Redacted Cartel codebase and in particular their new BTRFLY V2 implementation along with the surrounding migration infrastructure.

Over the course of the audit, we identified a potential misbehaviour within the `Mariposa` contract as well as several minor issues around standard conformity and potential deviances from the original BTRFLY V1 implementation.

We advise the Redacted Cartel 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.

Post-Audit Conclusion

The Redacted Cartel team produced a response document as well as detailed commits for each of the exhibits outlined as well as produced test suites meant to nullify one of the exhibits outlined in the report.

We advise the TMR-01M, MAR-01M, MAR-03C, and MAR-04C exhibits to be re-visited for further potential adjustments to the code prior to finalization.

Contracts Assessed

Files in Scope	Repository	Commit(s)
BTRFLYV2.sol (BTR)	v2-dev	8c2d2bb49d, 2e5dbfa109
Mariposa.sol (MAR)	v2-dev	8c2d2bb49d, 2e5dbfa109
RLBTRFLY.sol (RLB)	v2-dev	8c2d2bb49d, 2e5dbfa109
TokenMigrator.sol (TMR)	v2-dev	8c2d2bb49d, 2e5dbfa109

Audit Synopsis

Severity	Identified	Alleviated	Partially Alleviated	Acknowledged
Unknown	2	2	0	0
Informational	6	3	1	2
Minor	2	2	0	0
Medium	2	1	0	1
Major	0	0	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 **10 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 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 TypeScript.

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 `0.8.12` based on the version specified within the `hardhat.config.ts` file.

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

The contracts' `pragma` statements have been locked to `0.8.12` (`=0.8.12`), 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 **28 potential issues** within the codebase of which **23 were ruled out to be false positives** or negligible findings.

The remaining **5 issues** were validated and grouped and formalized into the **2 exhibits** that follow:

ID	Severity	Addressed	Title
TMR-01S	Minor	Yes	Potentially Unsafe Approve Invocations
TMR-02S	Medium	Yes	Improper Invocations of EIP-20 `transferFrom`

Manual Review

A **thorough line-by-line review** was conducted on the codebase to identify potential malfunctions and vulnerabilities in Redacted Cartel's new BTRFLY V2 token.

As the project at hand implements a token migration mechanism, intricate care was put into ensuring that the **flow of funds 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 race-condition** within the system's minting contract which could have had **moderate ramifications** to its overall operation, however, it was conveyed ahead of time to the Redacted Cartel team to be **promptly remediated**.

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 the extent it need be.

A total of **10 findings** were identified over the course of the manual review of which **4 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
BTR-01M	Unknown	Yes	Deviation From BTRFLY V1
MAR-01M	Medium	Acknowledged	Improper Mint Allowance Race Condition
RLB-01M	Minor	Yes	Unsafe Casting Operations
TMR-01M	Informational	Acknowledged	Favourable Conversion Rate

Code Style

During the manual portion of the audit, we identified **6 optimizations** that can be applied to the codebase that will decrease the gas-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
MAR-01C	Unknown	Yes	Misleading Contract Language
MAR-02C	Informational	Yes	Generic Typographic Mistake
MAR-03C	Informational	Partial	Inefficient Variable Reads
MAR-04C	Informational	No	Inefficient `mapping` Lookups
TMR-01C	Informational	Yes	First-Time Assignment Optimization
TMR-02C	Informational	Yes	`wxBTRFLY` Integration Optimization

TokenMigrator Static Analysis Findings

TMR-01S: Potentially Unsafe Approve Invocations

Type	Severity	Location
Standard Conformity	Minor	TokenMigrator.sol:L76, L77

Description:

The linked `approve` instructions may yield a `bool` variable that currently remains unsanitized. As per the **EIP-20** standard, **callers MUST NOT assume that `false` is never returned**.

Impact:

A potential failure case for the approvals will require a redeployment of the protocol to ensure that it has sufficient allowance to interact with the revenue-locked BTRFLY as well as the staking implementation's unstake function.

Example:

contracts/core/TokenMigrator.sol

SOL

```sol
76  xBtrfly.approve(staking_, type(uint256).max);
77  btrflyV2.approve(rlBtrfly_, type(uint256).max);
```

Recommendation:

We advise the `SafeTransferLib` implementation already utilized by the codebase to be imported here and its `safeApprove` function to be utilized. To note, the Solmate implementation does not suffer from the same flaws as OpenZeppelin's and as such is safe to use in this scenario as it solely evaluates the returned `bool`.

Alleviation:

The correct `safeApprove` implementation from the Solmate library is now utilized in the codebase as advised.

TMR-02S: Improper Invocations of EIP-20 `transferFrom`

Type	Severity	Location
Standard Conformity	Medium	TokenMigrator.sol:L90, L106, L121

Description:

The linked statements do 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/core/TokenMigrator.sol

SOL

```
90  wxBtrfly.transferFrom(msg.sender, address(this), amount);
```

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 in each instance.

Alleviation:

All relevant `transferFrom` invocations have been properly replaced by their OpenZeppelin `SafeERC20` counterparts alleviating this exhibit.

BTRFLYV2 Manual Review Findings

BTR-01M: Deviation From BTRFLY V1

Type	Severity	Location
Standard Conformity	Unknown	BTRFLYV2.sol:L15-L30

Description:

The new `BTRFLY` V2 token implementation lacks important functions that were present in the previous implementation, such as the `burn` and `burnFrom` functions that make the current V1 -> V2 migration possible.

Example:

contracts/core/BTRFLYV2.sol

SOL

```sol
contract BTRFLYV2 is AccessControl, ERC20("BTRFLY", "BTRFLY", 18) {
    bytes32 public constant MINTER_ROLE = "MINTER_ROLE";

    constructor() {
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    }

    /**
        @notice Mint tokens
        @param  to      address  Address to receive tokens
        @param  amount  uint256  Amount to mint
     */
    function mint(address to, uint256 amount) external onlyRole(MINTER_ROLE) {
        _mint(to, amount);
    }
}
```

Recommendation:

We advise the missing features to be carefully evaluated and potentially re-introduced depending on the business use case of `BTRFLYV2`. Additionally, we would like to note that the new `BTRFLYV2` token contains 18 decimals whilst the original token contained 9 decimals, a discrepancy which may case any previous implementations being upgraded to misbehave.

Alleviation:

The `burn` function that was present in the original implementation has been re-introduced permitting future upgrades to be seamless similarly to the V1 -> V2 conversion and thus alleviating this exhibit.

Mariposa Manual Review Findings

MAR-01M: Improper Mint Allowance Race Condition

Type	Severity	Location
Language Specific	Medium	Mariposa.sol:L125

Description:

The linked conditional evaluated for decreasing the allowance of a particular minted is harmful as it leads to a race condition whereby a malicious minter will detect the owner's action of decreasing their allowance and will consume an allowance equal to `mintAllowances[minted] - amount + 1` causing the decrease operation to fail fatally.

Impact:

It is currently possible for a malicious minter to sabotage any allowance decrease operation by carefully consuming their allowance with a higher gas fee.

Example:

contracts/core/Mariposa.sol

SOL

```sol
/**
    @notice Decrease allowance
    @param  minter  address  Address with minting rights
    @param  amount  uint256  Amount to decrease
 */
function decreaseAllowance(address minter, uint256 amount)
    external
    onlyOwner
{
    if (!isMinter[minter]) revert NotMinter();
    if (amount == 0) revert ZeroAmount();
    if (mintAllowances[minter] < amount) revert UnderflowAllowance();

    totalAllowances -= amount;
    mintAllowances[minter] -= amount;

    emit DecreasedAllowance(minter, amount);
}
```

Recommendation:

We advise the condition to be gracefully handled by assigning `mintAllowances[minter]` to `amount` if `amount` exceeds it, preventing the race condition from manifesting.

Alleviation:

The Redacted Cartel considered this exhibit and opted to not apply a remediation for it as the minter accounts will solely be managed by the team. We would like to note that the recommended alleviation mitigates a portion of errors that may surface during `decreaseAllowance` operations, such as an automatic smart contract that is meant to be taken off operation but is utilized by public users. As such, we advise the graceful handling of allowance decrease to be introduced into the codebase but consider this exhibit acknowledged as the team expressed desire to perform no action.

RLBTRFLY Manual Review Findings

RLB-01M: Unsafe Casting Operations

Type	Severity	Location
Mathematical Operations	Minor	RLBTRFLY.sol:L236, L252, L281

Description:

The linked statements perform unsafe casting operations to the `uint224` and `uint32` data types.

Impact:

An unsafe casting operation can cause the value cast to underflow and lead to the overall system misbehaving.

Example:

contracts/core/RLBTRFLY.sol

SOL

```sol
236 uint224 lockAmount = uint224(amount);
```

Recommendation:

Contrary to what one may expect, the built-in safe arithmetics of Solidity (`^0.8.X`) do not account for casting operations and thus these cases need to be manually handled. We advise the operations to be performed safely via utility functions that check the casted variables against the maximum of each respective type (i.e. `type(uint224).max`).

Alleviation:

Proper safe casting operations are now utilized by the code ensuring that casting overflows can no longer occur in the referenced statements.

TokenMigrator Manual Review Findings

TMR-01M: Favourable Conversion Rate

Type	Severity	Location
Mathematical Operations	Informational	TokenMigrator.sol:L148-L151

Description:

Due to the way the `wxBTRFLY` token works, a conversion from `(x)BTRFLY` to `wBTRFLY` (`wBTRFLYValue`) will truncate causing the `wBTRFLY` to `(x)BTYRFLY` conversion (`xBTRFLYValue`) to yield one unit less than the original input amount. Given that ultimately the `wBTRFLY` evaluation is utilized for minting the `BTRFLY` V2 token, it is favourable for users to wrap their tokens in `wxBTRFLY` and then convert them.

Example:

contracts/core/TokenMigrator.sol

SOL

```sol
148 if (wxAmount != 0) {
149     burnAmount += wxBtrfly.xBTRFLYValue(wxAmount);
150     mintAmount += _migrateWxBtrfly(wxAmount);
151 }
152
153 if (xAmount != 0) {
154     burnAmount += xAmount;
155     mintAmount += _migrateXBtrfly(xAmount);
156 }
157
158 if (v1Amount != 0) {
159     burnAmount += v1Amount;
160     mintAmount += _migrateBtrfly(v1Amount);
161 }
```

Recommendation:

While the degree to which users gain a benefit from this action is miniscule, we advise this behaviour to be documented via in-line comments warning that due to truncations, the same `BTRFLY` amount may yield different results depending on the form (normal, `x` and `wx`) utilized for the conversion.

Alleviation:

The Redacted Cartel team introduced a property-based test in their repository to invalidate this finding as truncation never affects the converted amounts based on the test. We would like to note that the `wBTRFLYValue` function of `wxBTRFLY` performs a division with the result of `realIndex`. This causes a truncation as long as the amount is equivalent to `n * realIndex() - 1` where `n` is a number greater-than-or-equal-to (`>=`) two. Please test and ensure that the above does not occur in which case we will nullify this finding.

Mariposa Code Style Findings

MAR-01C: Misleading Contract Language

Type	Severity	Location
Code Style	Unknown	Mariposa.sol:L65, L75, L77

Description:

The linked `request` function hints that a mint has been requested, however, the function directly fulfils the mint request by minting the relevant tokens to the intended `recipient`.

Example:

contracts/core/Mariposa.sol

SOL

```sol
/**
    @notice Mints tokens for recipient
    @param  recipient  address  To receive minted tokens
    @param  amount     uint256  Amount
 */
function request(address recipient, uint256 amount) external whenNotPaused {
    if (!isMinter[msg.sender]) revert NotMinter();
    if (amount == 0) revert ZeroAmount();
    if (recipient == address(0)) revert ZeroAddress();
    if (amount > mintAllowances[msg.sender]) revert ExceedsAllowance();

    emissions += amount;
    mintAllowances[msg.sender] -= amount;
    totalAllowances -= amount;

    emit Requested(msg.sender, recipient, amount);

    btrflyV2.mint(recipient, amount);
}
```

Recommendation:

We advise the language around the function and the corresponding event emitted within it to be changed as it currently is misleading.

Alleviation:

The `request` function and associated language have been properly adjusted to `mintFor` and similar language thereby increasing the code's clarity.

MAR-02C: Generic Typographic Mistake

Type	Severity	Location
Code Style	Informational	Mariposa.sol:L50

Description:

The referenced line contains a typographical mistake or generic documentational error (i.e. copy-paste) that should be corrected.

Example:

contracts/core/Mariposa.sol

SOL

```
50  @param  _supplyCap  uint256  Max number of tokens contract can emmit
```

Recommendation:

We advise this to be done so to enhance the legibility of the codebase.

Alleviation:

The typographic mistake has been corrected properly.

MAR-03C: Inefficient Variable Reads

Type	Severity	Location
Gas Optimization	Informational	Mariposa.sol:L105, L108

Description:

The linked statements read variables from `storage` redundantly twice instead of caching them to local variables.

Example:

contracts/core/Mariposa.sol

SOL

```sol
105 if (emissions + totalAllowances + amount > supplyCap)
106     revert ExceedsSupplyCap();
107
108 totalAllowances += amount;
```

Recommendation:

Given that the relevant variables are not meant to change between read operations, we advise each variable that is read repeatedly to be read once and stored to a local variable that is consequently utilized optimizing the gas cost of all linked statements.

Alleviation:

An optimization has been applied albeit not to the maximum level possible. The `t` variable should hold the result of `totalAllowances + amount` which is consequently evaluated in the `if` conditional (`emissions + t > supplyCap`) and ultimately assigned to the `totalAllowances` variable after the conditional has been evaluated properly.

MAR-04C: Inefficient `mapping` Lookups

Type	Severity	Location
Gas Optimization	Informational	Mariposa.sol:L69, L72, L125, L128

Description:

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

Example:

contracts/core/Mariposa.sol

SOL

```sol
69  if (amount > mintAllowances[msg.sender]) revert ExceedsAllowance();
70
71  emissions += amount;
72  mintAllowances[msg.sender] -= amount;
```

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:

The former of the two referenced variable read pairs is no longer present in the codebase whereas the latter still exists and has not been remediated. We advise the value of `mintAllowances[minter]` to be stored to a local variable that is utilized for the `if` conditional and used in the assignment to `mintAllowances[minter]` without a compound operator which indirectly reads the variable again.

TokenMigrator Code Style Findings

TMR-01C: First-Time Assignment Optimization

Type	Severity	Location
Gas Optimization	Informational	TokenMigrator.sol:L149-L150

Description:

The linked statements increment the `burnAmount` and `mintAmount` variables, however, it is known that in the first `if` clause they will always be zero thus permitting a direct assignment to be made instead.

Example:

contracts/core/TokenMigrator.sol

SOL

```sol
145 uint256 burnAmount;
146 uint256 mintAmount;
147
148 if (wxAmount != 0) {
149     burnAmount += wxBtrfly.xBTRFLYValue(wxAmount);
150     mintAmount += _migrateWxBtrfly(wxAmount);
151 }
```

Recommendation:

We advise this to be done so optimizing the gas cost of those statements.

Alleviation:

Both assignment operations were adjusted to simple assignments optimizing the codebase.

TMR-02C: `wxBTRFLY` Integration Optimization

Type	Severity	Location
Gas Optimization	Informational	TokenMigrator.sol:L91, L149-L150

Description:

The `unwrapToBTRFLY` function of `wxBTRFLY` yields the amount of `BTRFLY` that was ultimately unwrapped. In the `TokenMigrator` contract, this is measured in a separate call to `xBTRFLYValue` which is inefficient.

Example:

contracts/core/TokenMigrator.sol

SOL

```sol
function _migrateWxBtrfly(uint256 amount)
    internal
    returns (uint256 mintAmount)
{
    // Unwrap wxBTRFLY
    wxBtrfly.transferFrom(msg.sender, address(this), amount);
    wxBtrfly.unwrapToBTRFLY(amount);

    return amount;
}
```

Recommendation:

We advise the `_migrateWxBtrfly` function to yield the `burnAmount` instead and be assigned to it whilst the `mintAmount` is directly assigned to the `wxAmount`, optimizing the code's execution cost.

Alleviation:

The codebase was adjusted as advised optimizing the execution cost of the `wxBTRFLY` conversion path.

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.