Security Review Report
NM-0366 CCDM

NETHERMIND
SECURITY

(Dec 9, 2024)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security of the Cross-chain Deposit Module (CCDM). CCDM is a system that facilitates cross-chain deposit campaigns to provide liquidity to protocols on pre-launch or existing chains. Incentive Providers (IPs) create CCDM enabled Royco markets on the source chain to incentivize Action Providers (APs) to provide liquidity to protocols on a destination chain. CCDM taps into idle or lower-yielding liquidity on other chains to solve the cold-start problem for new protocols and increase liquidity reserves for existing protocols, benefiting the chain and its protocols. A single CCDM instance can facilitate deposit campaigns for a single source-destination chain pair.

The audited code comprises 797 lines of code written in the Solidity language. The audit focused on the implementation of the CCDM system.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases. **Along this document, we report** eleven points of attention, three of which are classified as `Critical` severity, one is classified as `High` severity, two are classified as `Medium` severity and five are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation. Section 9 concludes the document.

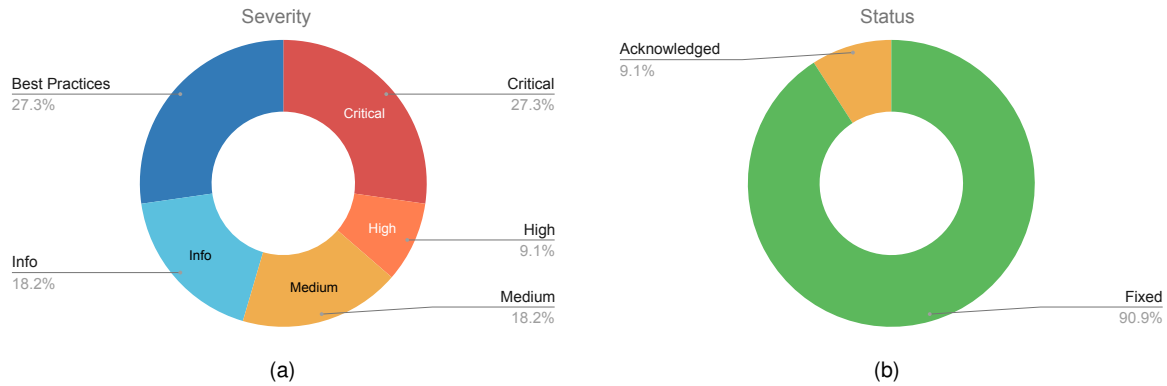

Fig. 1: Distribution of issues: Critical (3), **High** (1), **Medium** (2), **Low** (0), **Undetermined** (0), **Informational** (2), **Best Practices** (3). **Distribution of status: Fixed** (10), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	December 6, 2024
Final Report	December 9, 2024
Repository	cross-chain-deposit-module
Commit	98d248d3e16daf8397efbb83253ee10108a12e41
Final Commit	da032b0474feef05c05e23dea5ca77bc2c11ebf2
Documentation	Docs
Documentation Assessment	High
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/core/DepositLocker.sol	468	369	78.8%	143	980
2	src/core/DepositExecutor.sol	276	267	96.7%	97	640
3	src/libraries/CCDMPayloadLib.sol	53	43	81.1%	11	107
	Total	**797**	**679**	**85.2%**	**251**	**1727**

3 Summary of Issues

	Finding	Severity	Update
1	Campaign owner can steal tokens bridged for other campaigns	Critical	Fixed
2	Depositing into a non-existent market allows funds to be stolen when the market is later created	Critical	Fixed
3	The `lzCompose(...)` function can be provided with arbitrary input to create artificial deposits and steal other users funds	Critical	Fixed
4	If `WeirollWallet` is executed before all the assets are bridged, tokens will be permanently stuck	High	Fixed
5	Bridged tokens can be stuck permanently if the campaign's input tokens are changed after bridging but before the wallet's execution	Medium	Fixed
6	Token bridging can be griefed	Medium	Fixed
7	Bridged deposits that revert with out of gas error, must be re-tried to start participating in the campaign	Info	Acknowledged
8	Dust amounts left in Weiroll Wallet after executing the deposit recipes are not explicitly handled	Info	Fixed
9	Weiroll Wallet can have an unlock timestamp of zero if funds were bridged before campaign params were set	Best Practices	Fixed
10	Missing event emission in `constructor(...)`	Best Practices	Fixed
11	Missing input validation in `_setLzV2OFTForToken` when `_token` is a chain's native asset	Best Practices	Fixed

4 System Overview

The Cross-Chain Deposit Module (CCDM) is a sophisticated system built on Royco that is designed to facilitate cross-chain deposit campaigns. Incentive Providers (IPs) can incentivize Action Providers (APs) to commit liquidity on a source chain towards agreed-upon actions (supplying, LPing, swapping, etc.) on a destination chain. Royco provides efficient price discovery for protocols trying to acquire liquidity for a desired timeframe. CCDM handles safely transporting this liquidity to the intended protocol in a trust-minimized manner. A single CCDM instance can facilitate cross-chain deposit campaigns from a single source chain to a single destination chain. CCDM consists of two core components: the Deposit Locker on the source chain and the Deposit Executor on the destination chain. All incentive negotiation and liquidity provision is powered by Royco Recipe Incentivized Action Markets (IAMs) that live on the source chain.

The diagram below showcases a high-level view of the system's architecture. The following sections delve into the system's components and their interactions.

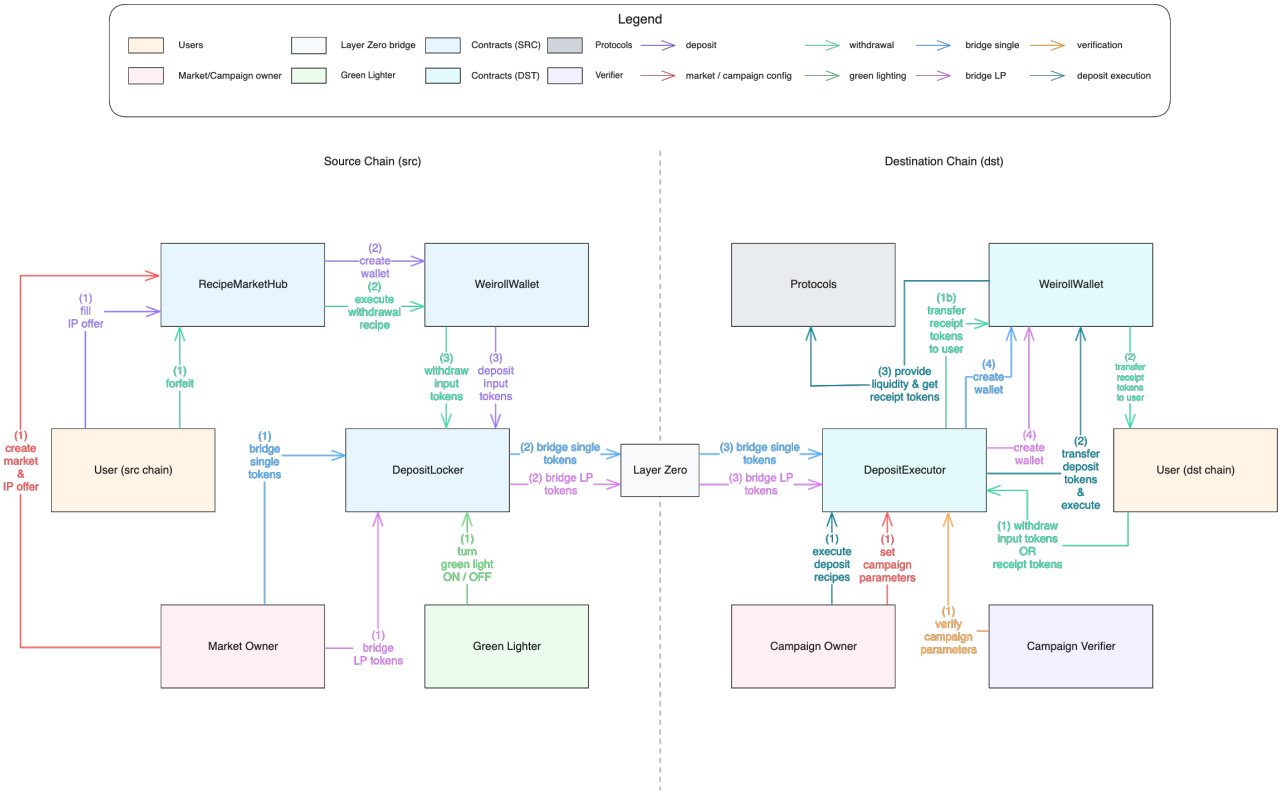

Fig. 2: CCDM - Structural Diagram of contracts. Note that the diagram does not present the flow of deposit locker configuration.

4.1 Actors

- **Deposit Locker and Executor owner**: The owner of the `DepositLocker` and `DepositExecutor` contracts controls the key parameters of the CCDM system. The party that deployed the contracts to support a new chain is assigned as the first owner. The holder of this role can set and remove other roles, such as campaign/market owners, green lighters, and campaign verifiers.
- **Incentive Providers**: Protocols looking for liquidity. They create markets in which users can provide liquidity in return for incentives. They control the deposit campaign settings, such as the accepted input tokens, the deposit recipe, the campaign's token unlock timestamp, and offered receipt tokens.
- **Action Providers**: Users (Depositors) provide liquidity to Incentive Providers in return for incentives.
- **Campaign Verifier**: Third party responsible for verifying deposit recipes on the destination chain. Verification also includes validating whether the deposit input and receipt tokens used by the recipe were set correctly by the campaign owner.
- **Green Lighter**: A role that monitors the CCDM system and marks users' deposits as bridgeable if all the other parties fulfilled their duties correctly.

4.2 Depositing assets

Incentive Providers (IPs) offer incentives for Action Providers (APs) to provide liquidity, which will be bridged to protocols launching on new chains. When APs fulfill an IP offer on a CCDM-enabled market, a new Weiroll Wallet will be created that will call the `deposit(...)` function in the `Deposit Locker` contract. `ERC20` tokens accepted by the `Deposit Locker` have to be whitelisted by the contrac's owner. The requirement is that the token has to have the corresponding OFT contract. In the case of Uniswap V2 LP tokens, each constituent have to have a matching OFT. The deposited assets will be mapped to the market hash of the CCDM-enabled market and will be bridged after the green light is given for the market.

4.3 Bridging of assets

Before the assets can be bridged to the destination chain, Green Lighter has to assess the state of the two chains. Once that's done, a green light will be given, and a rage quit period of 48 hours will start. It gives users a final chance to withdraw the funds before they become bridgeable. After 48 hours, anyone can bridge tokens for a single asset market by calling `bridgeSingleTokens(...)` function. In the case of LP token markets, bridging must be initiated by the market owner's call to `bridgeLpTokens(...)` function. The cross-chain asset transfer is facilitated via LayerZero's Omnichain Fungible Token (OFT) standard. Each bridging operation is tied to a `ccdm` nonce. When the cross-chain message is received by the `DepositExecutor` contract on the destination chain, a Weiroll Wallet is created for each nonce. LP token constituents are assigned the same nonce but are bridged in two separate transactions. Once bridging is complete, the campaign owner can execute the deposit recipes to use the provided liquidity.

4.4 Recipe execution

If the verifier confirms the campaign's parameters, the campaign is considered verified and is ready to be executed. The campaign's owner calls the `executeDepositRecipes(...)` function in the `DepositExecutor` contract and provides the list of `Weiroll Wallets` to be executed with the verified deposit recipe. Funds are pulled from the `DepositExecutor` contract into individual `Weiroll Wallets`, and the recipe is executed. The steps performed inside the recipe will vary based on the campaign. The common factor among all campaigns, is that the wallet must increase its balance of the campaign's receipt token after the recipe execution as a sign of provided liquidity. The receipt tokens can be later withdrawn from the wallets by the depositors after the unlock time for the campaign has passed.

4.5 Withdrawal of assets

Users can withdraw the assets; however, the withdrawal process will differ depending on the state of the CCDM system. If the assets have not been bridged yet, they can be withdrawn from the `DepositLocker` contract in two scenarios:

a. For forfeitable Incentive Action Markets (IAMs), depositors can forfeit their incentives by calling the `forfeit(...)` function in `Recipe Market Hub`. If a withdrawal recipe is specified for the market, it will automatically call the `withdraw(...)` in the `Deposit Locker` and return the funds to the user. Depositors can also call `manualExecuteWeiroll(...)` or `execute(...)` in their `Weiroll Wallet` with proper arguments to call `withdraw(...)` manually.

b. In rare cases where the wallet's unlock timestamp passed, but the funds have not been bridged, depositors can follow similar steps to the ones outlined above, but without forfeiting any incentives.

If the funds were bridged, the withdrawal has to be performed via `DepositExecutor` contract on the destination chain.

The depositors can withdraw their receipt tokens if the campaign owner has successfully executed the deposit recipe. Otherwise, they can withdraw the originally bridged tokens. Both operations are facilitated via the `withdraw(...)` function in the `Deposit Executor` contract.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] Campaign owner can steal tokens bridged for other campaigns

File(s): src/core/DepositExecutor.sol

Description: Right before a particular Weiroll Wallet is executed in the `executeDepositRecipes(...)` function, the campaign's input tokens are transferred from the `DepositExecutor` contract to the wallet's address. This ensures that the deposit recipe using the wallet can access relevant tokens. The amount of tokens sent to the wallet equals what was deposited by the user in the `DepositLocker` and later sent and processed in the `DepositExecutor`'s `lzCompose(...)` function. This value is stored in Weiroll Wallet's accounting `tokenToTotalAmountDeposited` mapping.

An issue arises if, before calling the `executeDepositRecipes(...)` function, the campaign owner modifies the campaign's input tokens to contain duplicated entries. In that case, the `_transferInputTokensToWeirollWallet(...)` function would iterate over an `_inputTokens` array and send the user's deposit multiple times to a single Weiroll Wallet.

```
function _transferInputTokensToWeirollWallet(
    ERC20[] storage _inputTokens,
    WeirollWalletAccounting storage _walletAccounting,
    address _weirollWallet
)
    internal
{
    // @audit For each input token, get the user's deposit and send it to the wallet.
    for (uint256 i = 0; i < _inputTokens.length; ++i) {
        // @audit Fhe _inputTokens array can contain duplicated entries.
        ERC20 inputToken = _inputTokens[i];

        // @audit The deposited amount has not been decreased/modified; it stays the same.
        uint256 amountOfTokenDepositedIntoWallet =
            _walletAccounting.tokenToTotalAmountDeposited[inputToken];

        // @audit As a result, the deposit amount will be transferred multiple times.
        inputToken.safeTransfer(
            _weirollWallet, amountOfTokenDepositedIntoWallet
        );
    }
}
```

In subsequent loop iterations, the tokens sent to the wallet might come from other deposit campaigns and belong to other users. Malicious campaign owners can exploit this mechanism by depositing to their campaign, adding many duplicated entries to the input tokens array, and executing the deposit recipes, effectively stealing tokens from other campaigns.

Recommendation(s): Consider removing the `setCampaignInputTokens(...)` function, thus, remove the ability of campaign owner to modify the input tokens. Rather, input tokens can be set in `lzCompose(...)` function since they were transferred for that particular market hash after confirming them in the `RecipeMarketHub` contract on the source chain. Enumerable Set from OpenZeppelin can be used to make sure that the token is a member of the set, that it does not contain duplicate elements, and it can be easily iterated.

Status: Fixed.

Update from the client: 81d8a0e

6.2 [Critical] Depositing into a non-existent market allows funds to be stolen when the market is later created

File(s): src/core/DepositLocker.sol

Description: Whenever the user fills an offer, the `deposit(...)` function on the `DepositLocker` contract transfers the market's input token from the user to the locker contract. The token to be used is queried from the `RecipeMarketHub` contract, based on the `marketHash` associated with the Weiroll Wallet performing the deposit. After that, the token is taken from the user, and internal accounting is updated to reflect the user's deposit.

The problem present in the `deposit(...)` function is that it does not validate the queried token's address. The mapping that associates token addresses with market hashes will return an invalid token address in two cases:

- The market was created for a token that is not yet deployed (e.g., a future UniV2 Pair);
- The market with that particular market hash does not exist;

In both scenarios, the `safeTransferFrom(...)` function from the `Solmate` library will be called on an address that is not a token contract. This poses an issue since `Solmate`'s `SafeTransferLib` uses inline assembly and does not check for code existence at the token address. This responsibility is delegated to the caller. The token transfer wouldn't revert in the scenarios outlined above even though no tokens were sent. The internal accounting would still be updated in the `DepositLocker` contract.

```
function deposit() external nonReentrant {
    WeirollWallet wallet = WeirollWallet(payable(msg.sender));
    bytes32 targetMarketHash = wallet.marketHash();
    address depositor = wallet.owner();
    uint256 amountDeposited = wallet.amount();

    // @audit The market input token can be address 0 or have no code.
    (, ERC20 marketInputToken,,,,) =
        RECIPE_MARKET_HUB.marketHashToWeirollMarket(targetMarketHash);
    // ...

    // @audit Solmate's safeTransferFrom won't revert.
    marketInputToken.safeTransferFrom(
        msg.sender, address(this), amountDeposited
    );

    // @audit-issue Accounting is updated, which allows the attacker to withdraw later.
    marketHashToDepositorToAmountDeposited[targetMarketHash][depositor] +=
        amountDeposited;
    marketHashToDepositorToWeirollWallets[targetMarketHash][depositor].push(
        msg.sender
    );
    depositorToWeirollWalletToAmount[depositor][msg.sender] =
        amountDeposited;
    // ...
}
```

Once the market or the token pair is eventually created and legitimate users make deposits, the attacker can call `withdraw(...)` to steal the funds from the `DepositLocker` contract.

A malicious user can perform this attack by monitoring and front-running `createMarket(...)` transactions on the `RecipeMarketHub` contract by depositing to a soon-to-be-created `marketHash`. The same can be done for markets that anticipate the creation of a Uniswap V2 pair.

Recommendation(s): Consider adding a code size check for `marketInputToken` to ensure that artificial token transfers are not possible.

Status: Fixed.

Update from the client: e484c55

6.3 [Critical] The `lzCompose(...)` function can be provided with arbitrary input to create artificial deposits and steal other users funds

File(s): src/core/DepositExecutor.sol

Description: Stargate is an oApp built on top of LayerZero's cross-chain messaging system. To communicate between two chains, it defines the `lzSend(...)` function on the source chain and `lzReceive(...)` on the destination chain. These two functions form the basis of cross-chain communication. The LayerZero's V2 upgrade introduced the composability feature called `lzCompose(...)`, which allows oApps to pack an arbitrary external call and send it along the standard send-and-receive message. On the destination chain, the calls to `lzReceive(...)` and `lzCompose(...)` are executed independently in two separate transactions.

The `lzReceive(...)` function defined in the Stargate's `TokenMessaging` contract is called first. It calls Stargate's Pool contract to perform the token transfers and inform the LZ endpoint about the incoming compose call. This is done by calling the `sendCompose(...)` function of the `EndpointV2` contract.

```
// EndpointV2.sol
function sendCompose(
    address _to,
    bytes32 _guid,
    uint16 _index,
    bytes calldata _message
)
    external
{
    // @audit sendCompose is permissionless to allow any oApp
    // use the composability features.
    if (composeQueue[msg.sender][_to][_guid][_index] != NO_MESSAGE_HASH) {
        revert Errors.LZ_ComposeExists();
    }
    composeQueue[msg.sender][_to][_guid][_index] = keccak256(_message);
    emit ComposeSent(msg.sender, _to, _guid, _index, _message);
}
```

The `sendCompose(...)` function is permissionless and allows any oApp to store the message to be executed in `lzCompose(...)`. The entry in the `composeQueue` is tied to the destination chain's receiver contract that called `sendCompose(...)`. In the case of Stargate, this is the Stargate's Pool.

The `lzCompose(...)` can be called right after that. This function is also permissionless but is often called by LayerZero's executor bot. As long as the data passed to this function matches what was previously stored in `sendCompose(...)`, the target (`DepositExecutor` contract) will be called.

```
// EndpointV2.sol
function lzCompose(
    address _from,
    address _to,
    bytes32 _guid,
    uint16 _index,
    bytes calldata _message,
    bytes calldata _extraData
)
    external
    payable
{
    // @audit check that hash was stored by calling sendCompose earlier
    bytes32 expectedHash = composeQueue[_from][_to][_guid][_index];
    bytes32 actualHash = keccak256(_message);
    if (expectedHash != actualHash) {
        revert Errors.LZ_ComposeNotFound(expectedHash, actualHash);
    }

    composeQueue[_from][_to][_guid][_index] = RECEIVED_MESSAGE_HASH;

    // @audit Call the target e.g. DepositExecutor
    ILayerZeroComposer(_to).lzCompose{ value: msg.value }(
        _from, _guid, _message, msg.sender, _extraData
    );
}
```

The inherent risk of this permissionless design is that the target of the external call must validate who stored the message in the `composeQueue`. This is not the case in the `DepositExecutor`'s `lzCompose(...)` function.

```sol
// DepositExecutor.sol
function lzCompose(
    address _from,
    bytes32 _guid,
    bytes calldata _message,
    ...
)
    external
    payable
    nonReentrant
{
    // @audit endpoint will call this function as a result of a permissionless
    // call to lzCompose on the endpoint. The check will pass.
    require(msg.sender == LAYER_ZERO_V2_ENDPOINT, NotFromLzV2Endpoint());

    // @audit-issue The attacker controls the compose msg via sendCompose(...)
    bytes memory composeMsg = OFTComposeMsgCodec.composeMsg(_message);

    // @audit-issue The amountLD is normally controlled by Stargate,
    // but the attacker can manipulate it as well.
    uint256 tokenAmountBridged = OFTComposeMsgCodec.amountLD(_message);

    // @audit Data from the _message is used in the internal accounting,
    // but is fully controlled by the attacker.

    // ...
}
```

By controlling the message, the attacker does not have to deposit any campaign tokens, but they will be credited to him in the campaign's `weirollWalletToAccounting` mapping. Once the campaign's owner executes the deposit recipes, the attacker's weiroll wallet will receive the receipt tokens in exchange for liquidity that was provided by other users.

Recommendation(s): When receiving a composable LayerZero message, three things should be checked by the `DepositExecutor` contract:

- Checking that the current caller is LZ Endpoint;
- Checking that the compose message was registered by an authorized OFT;
- Checking that `DepositLocker` sent the message on the source chain;

The first check is already implemented. To resolve the issue, consider implementing the two missing checks. The `_from` address of `lzCompose(...)` can be used to handle the 2nd check, e.g., `_from == Stargate Pool` similarly in which `tokenToLzV2OFT` holds authorized OFTs in the Locker contract. For the 3rd check, the address of the `DepositLocker` can be retrieved by using the `composeFrom(...)` function from the `OFTComposeMsgCodec`.

Status: Fixed.

Update from the client: b9e2dfc

6.4 [High] If `WeirollWallet` is executed before all the assets are bridged, tokens will be permanently stuck

File(s): src/core/DepositExecutor.sol

Description: Whenever the LP tokens are bridged from the `DepositLocker` contract, they are sent as two separate bridging transactions but are assigned the same `ccdm` nonce. The nonce is tied to a single Weiroll Wallet on the destination chain. Once the funds are bridged, the `executeDepositRecipes(...)` function can be called to execute the deposit recipe and use the liquidity provided by the user. Once that's done, the wallet is marked as executed, and subsequent executions of the same wallet are not possible. An issue might arise if the `executeDepositRecipes(...)` function happens to be called in between the two bridging transactions of the LP token pair constituents. The token that arrived first will be used in the deposit recipe. Since liquidity was provided successfully, this operation will yield a certain amount of receipt tokens. Once the second bridging transaction is finished, the Weiroll Wallet tied to the `ccdm` nonce that was just bridged is already executed. The second token can't be used to provide liquidity and will not yield receipt tokens.

The problem is that these tokens are permanently stuck in the `DepositExecutor` contract. They can't be withdrawn directly since the wallet is considered `executed`. At the same time, they can't be withdrawn as receipt tokens since they were not used in a deposit recipe and internal accounting was not updated. This scenario can happen by accident on every LP token bridge or on purpose if the campaign owner is malicious or compromised.

Recommendation(s): Consider adding a field in compose message received in `lzCompose(...)` function to know the number of tokens bridged and associated with the same `ccdm` nonce. So untill and unless all the tokens are bridged i.e. in case of LP tokens, both the tokens are bridged, `executeDepositRecipes(...)` function shouldn't be allowed to be called.

Status: Fixed.

Update from the client: 98d2bd9

6.5 [Medium] Bridged tokens can be stuck permanently if the campaign's input tokens are changed after bridging but before the wallet's execution

File(s): src/core/DepositExecutor.sol

Description: Once the Green Lighter validates the campaign's action logic and parameters, a green light is given, and the funds can be bridged from the `DepositLocker` to the `DepositExecutor` contract. After the `DepositExecutor` processes the bridging transactions, the created Weiroll Wallets can be executed with deposit recipe actions to start the liquidity provision campaign. An issue might arise if the campaign owner is compromised or acts maliciously by removing one of the previously configured input tokens for this campaign. If the wallet is executed with a single input token configured, even though two were previously bridged, the second token will be stuck for the same reason outlined in *[High] If* `WeirollWallet` *is executed before all the assets are bridged, tokens will be permanently stuck* finding. This can happen even if the Green Lighter fulfills its duties correctly, as certain campaign parameters, such as input tokens, can be changed after the green light is given (e.g., removing one of the two input tokens).

Recommendation(s): Consider removing the `setCampaignInputTokens(...)` function, thus, remove the ability of campaign owner to modify the input tokens. Rather, input tokens can be set in `lzCompose(...)` function since they were transferred for that particular market hash after confirming them in the `RecipeMarketHub` contract on the source chain. Enumerable Set from OpenZeppelin can be used to make sure that the token is a member of the set, that it does not contain duplicate elements, and it can be easily iterated.

Status: Fixed.

Update from the client: 81d8a0e

6.6 [Medium] Token bridging can be griefed

File(s): src/core/DepositLocker.sol

Description: Whenever the user deposits funds into the `DepositLocker` contract, the address of the Weiroll Wallet used in this interaction is pushed as an entry in the `marketHashToDepositorToWeirollWallets` array.

While bridging tokens from the source chain to the destination chain, the `_clearDepositorData(...)` function is used to clear the depositor's data and the abovementioned array. The array will be iterated in this function to clear the data entries.

Since this iteration is unbounded, a problem might arise. Any malicious contract can act as a Weiroll Wallet and call `deposit(...)` on behalf of any user with a very small amount (like one wei) multiple times to increase the array's length. This will lead to an out-of-gas error when the `_clearDepositorData(...)` function loops through the array, preventing a particular user's funds from being bridged.

```
function deposit() external nonReentrant {
    // @audit Any malicious contract can also act as a Weiroll Wallet
    WeirollWallet wallet = WeirollWallet(payable(msg.sender));
    bytes32 targetMarketHash = wallet.marketHash();
    address depositor = wallet.owner();
    uint256 amountDeposited = wallet.amount();

    // ...
    marketHashToDepositorToAmountDeposited[targetMarketHash][depositor] +=
        amountDeposited;

    // @audit Calling this function would increase the length of the array
    marketHashToDepositorToWeirollWallets[targetMarketHash][depositor].push(
        msg.sender
    );
    depositorToWeirollWalletToAmount[depositor][msg.sender] =
        amountDeposited;
    // ...
}

function _clearDepositorData(
    bytes32 _marketHash,
    address _depositor
)
    internal
{
    address[] storage depositorWeirollWallets =
        marketHashToDepositorToWeirollWallets[_marketHash][_depositor];

    // @audit-issue Looping through the large array can lead to out of gas error
    for (uint256 i = 0; i < depositorWeirollWallets.length; ++i) {
        // Set the amount deposited by the Weiroll Wallet to zero
        delete depositorToWeirollWalletToAmount[_depositor][
        depositorWeirollWallets[i]
    ];
    }
    // ...
}
```

Although the user (victim) can withdraw their funds at any point, any token deposit linked with the user's address can't be bridged due to the out-of-gas issue. The user would be forced to deposit from a different account. Competitors of protocols are incentivized to carry out this attack on whales, providing liquidity to the protocol to discourage them from doing so.

Recommendation(s): Consider changing the accounting variables for deposits. The array corresponding to `marketHashToDepositorToWeirollWallets` mapping can be removed. Instead, the nonce can be used which increments after tokens are bridged. Thus, keeping an array of all the deposits won't be needed.

Status: Fixed.

Update from the client: a1811e5, 9bbb29c

Update from Nethermind Security: In the `_processSingleTokenDepositor(...)` function the bridging nonce is incremented even in the case when deposit amount exceeds the `uint96` and no actual tokens are bridged. It makes it impossible to withdraw the funds then because the nonce was increased.

Update from Nethermind Security: Fixed.

6.7 [Info] Bridged deposits that revert with out of gas error, must be re-tried to start participating in the campaign

File(s): src/core/DepositLocker.sol

Description: The permissionless nature of the `bridgeSingleTokens(...)` allows anyone to to bridge the deposits of any depositor to the destination chain. It allows market owners to select deposits that they wish to bridge to to their protocol. The caller of the `bridgeSingleTokens(...)` provides the gas to be used to process the deposits on the destination chain. If either on purpose or by accident the caller specifies insufficient gas, the transaction will revert on the destination chain and must be retried by interacting with the destination chain's endpoint contract.

It is by design and in the best interest of market owners to monitor these transactions and re-try them if necessary. However, if that does not happen, the users might be forced to do it on their own which might cause inconvenience if they lack gas tokens on the destination chain.

Recommendation(s): Consider adding documentation around this edge case for the users to understand the conditions to which they agree when using CCDM.

Status: Acknowledged.

6.8 [Info] Dust amounts left in Weiroll Wallet after executing the deposit recipes are not explicitly handled

File(s): src/core/DepositExecutor.sol

Description: In the `executeDepositRecipes(...)` function, the deposit recipe is executed in the Weiroll Wallet containing the deposited tokens. In some cases, after executing the recipe there might be a dust amount of input tokens left in the Weiroll Wallet, that is not transferred back to the user. The deposit recipe may contain logic to handle that case, but there's no logic in the `DepositExecutor` to guarantee that depositors receive dust amounts of input tokens back.

Recommendation(s): Consider returning the unspent tokens to the user in the `withdraw(...)` function.

Status: Fixed.

Update from the client: PR 6

6.9 [Best Practice] Weiroll Wallet can have an unlock timestamp of zero if funds were bridged before campaign params were set

File(s): src/core/DepositExecutor.sol

Description: In the `lzCompose(...)` function, Weiroll Wallet is created if it wasn't created previously for a `ccdm` nonce. It is possible, however, that the bridging transaction was processed before the campaign parameters were set. If that's the case, a wallet might be created with an unlock timestamp of `0`. Some users might be able to withdraw the tokens. Still, the campaign owner can then modify the unlock timestamp by calling the `setCampaignUnlockTimestamp(...)` function, which will make the experience inconsistent for other users as they won't be able to withdraw instantly.

Recommendation(s): Consider validating that the campaign parameters were set before giving the green light.

Status: Fixed.

Update from the client: This makes unlock timestamp immutable after it is set once or the first batch of deposits is received: 81d8a0e

Although this doesn't stop a campaign owner from manipulating the unlock timestamp in between the first batch of deposits being bridged from the Locker and received by the Executor, it does keep the unlock timestamp consistent for all depositors in a campaign. The IP/Campaign owner's reputation will be tarnished if they choose to change the terms of the agreement before depositors have a chance to exit on the source chain. To mitigate the likelihood of this attack, if the campaign unlock timestamp is unset on the destination, Depositors should be notified of the risk on any CCDM interface. If campaign's intend on not having a lockup period, they should set their unlock timestamp to any non-zero timestamp less than the current timestamp.

6.10 [Best Practices] Missing event emission in `constructor(...)`

File(s): src/core/DepositLocker.sol, src/core/DepositExecutor.sol

Description: During the project's deployment, it is worth emitting events with the initial values for important protocol parameters. These make tracking on-chain issues easier in case they arise post-deployment. No such events are emitted:

- In the `constructor(...)` of the `DepositLocker` contract for `GREEN_LIGHTER`, `dstChainLzEid`, and `depositExecutor` state variables;
- In the `constructor(...)` of the `DepositExecutor` contract for the `campaignVerifier` address;

Such events are already emitted in the respective setter functions.

Recommendation(s): Consider emitting events with initial values in the `constructor(...)`.

Status: Fixed.

Update from the client: 5c6adc9

6.11 [Best Practices] Missing input validation in `_setLzV2OFTForToken` when `_token` is a chain's native asset

File(s): src/core/DepositLocker.sol

Description: The `_setLzV2OFTForToken(...)` function is used to set the LayerZero V2 OFT for a given token in the `tokenToLzV2OFT` mapping. When `_lzV2OFT` argument passed to the function corresponds to a native asset, i.e., `_lzV2OFT.token() == address(0)`, the `_token` argument isn't validated.

In this case, the `_token` should equal `WRAPPED_NATIVE_ASSET_TOKEN`. If `_setLzV2OFTForToken(...)` is called with a different `_token` equal to something other than `WRAPPED_NATIVE_ASSET_TOKEN`, the tx wouldn't fail, leading to undefined behavior later, especially while bridging the tokens.

```
function _setLzV2OFTForToken(ERC20 _token, IOFT _lzV2OFT) internal {

    address underlyingToken = _lzV2OFT.token();

    // @audit When underlyingToken == address(0),
    // no validation is done for address(_token)
    require(
        underlyingToken == address(_token) || underlyingToken == address(0),
        InvalidLzV2OFTForToken()
    );
    tokenToLzV2OFT[_token] = _lzV2OFT;
    // ...
}
```

Recommendation(s): Consider adding a check to ensure that the `_token` passed in `_setLzV2OFTForToken(...)` is equal to `WRAPPED_NATIVE_ASSET_TOKEN` if the `underlyingToken` is equal to `address(0)`.

Status: Fixed.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the CCDM documentation

The documentation for the CCDM contracts was provided through their official docs as well as detailed code comments. This documentation provided a high-level overview of the protocol and details of its implementation. Moreover, the CCDM team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Tests Output

```
forge test
[] Compiling...
[] Compiling 71 files with Solc 0.8.27
[] Solc 0.8.27 finished in 25.92s
Compiler run successful with warnings:
Warning (2018): Function state mutability can be restricted to view
   --> test/TestDepositLocker.t.sol:206:5:
    |
206 |     function assertDepositorState(address ap, address weirollWallet, uint256 fillAmount, uint256 filledSoFar)
 ↪  internal {
    |     ^ (Relevant source part starts here and spans across multiple lines).


Ran 2 tests for test/TestDepositLocker.t.sol:Test_DepositsAndWithdrawals_DepositLocker
[PASS] test_Deposits(uint256,uint256) (runs: 256, : 29216016, ~: 29158568)
[PASS] test_Withdrawals(uint256,uint256,uint256) (runs: 256, : 30620482, ~: 30730901)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 119.43s (224.26s CPU time)

Ran 2 tests for test/TestDepositExecutor.t.sol:E2E_Test_DepositExecutor
[PASS] test_ExecutorOnBridge_NoDepositRecipeExecution(uint256,uint256,uint256) (runs: 256, : 50429651, ~: 50212266)
[PASS] test_ExecutorOnBridge_WithDepositRecipeExecution(uint256,uint256,uint256) (runs: 256, : 53270704, ~: 52308220)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 151.16s (276.71s CPU time)

Ran 4 tests for test/TestBridgeDeposits.t.sol:Test_BridgeDeposits_DepositLocker
[PASS] test_Bridge_LpToken_wETH_USDC_Deposits(uint256,uint256,uint256) (runs: 256, : 47295626, ~: 47425765)
[PASS] test_Bridge_USDC_Deposits(uint256,uint256) (runs: 256, : 43018302, ~: 42237802)
[PASS] test_Bridge_wBTC_Deposits(uint256,uint256) (runs: 256, : 41669788, ~: 41861619)
[PASS] test_Bridge_wETH_Deposits(uint256,uint256) (runs: 256, : 40974617, ~: 39821082)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 177.77s (532.04s CPU time)

Ran 3 test suites in 177.78s (448.36s CPU time): 8 tests passed, 0 failed, 0 skipped (8 total tests)
```

Remarks about the CCDM contract's test suite

The test suite of the CCDM contracts covers all major flows of the protocol. However, upon closer review, it was observed that the suite lacks the necessary depth to validate certain edge cases and specific state changes. This limitation makes the test suite prone to errors, such as off-by-one issues, which can compromise the reliability of the protocol in critical scenarios.
In particular, the absence of targeted assertions to verify state transitions after each operation reduces the effectiveness of the tests in identifying subtle bugs. Enhancing the granularity and precision of the assertions could significantly improve the suite's resilience.
To address these gaps, incorporating mutation testing could be an effective approach. Mutation testing introduces small changes to the contract code to verify whether the test suite is capable of detecting them. This method can highlight untested or weakly tested paths in the code, providing a clear roadmap for improving test coverage.

8.2 Automated Tools

8.2.1 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE 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 OTHER ADVICE.