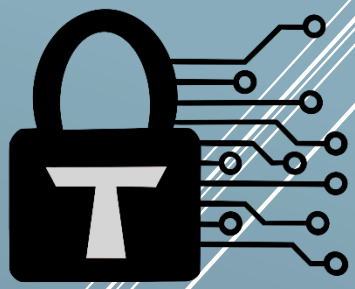
Trust Security



Smart Contract Audit

Mozaic.Fi Theseus Vault

13/01/2023

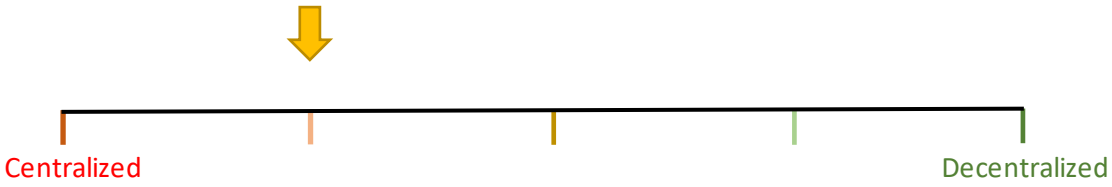
Executive summary

Category	Staking
Audited file count	4
Lines of Code	1098
Auditor	MiloTruck
Time period	11/12 – 20/12

Findings

Severity	Total	Fixed	Acknowledged	Open
High	5	5	0	0
Medium	9	8	1	0
Low	2	2	0	0

Centralization score



Signature

Digitally signed by Trust
Date: 2024.01.13 12:44:42 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	20/12/23	Client report
0.2	11/01/24	Mitigation review
0.3	13/01/24	Mitigation review #2

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- contracts/GmxPlugin.sol
- contracts/TokenPriceConsumer.sol
- contracts/Vault.sol
- contracts/GmxCallback.sol

Repository details

Thesus-vault:

- **Repository URL:** https://github.com/Mozaic-fi/Theseus-vault
- **Commit hash:** 2250955343202547af184a7f1c2ec1e80bafa69e
- **Mitigation review commit hash:** fcc2a22710d5d8cc3247417d53897dfc3b8685a4
- **Mitigation review #2 commit hash:** e5e287af5274bf2db9e943ea947272f9b33a40f5

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

MiloTruck is a blockchain security researcher who specializes in smart contract security. Since March 2022, he has competed in over 25 auditing contests on Code4rena and won several of them against the best auditors in the field. He has also found multiple critical bugs in live protocols on Immunefi and is an active judge on Code4rena.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	Moderate	Project is not complex, but some code could have been simplified.
Documentation	Mediocre	Project currently has limited documentation.
Best practices	Good	Project consistently adheres to industry standards.
Centralization risks	Moderate	Project has some centralization risks.

Findings

High severity findings

TRST-H-1: Withdrawals in *settleWithdrawRequest()* can be permanently DOSed

- **Category:** DOS attacks
- **Source:** Vault.sol
- **Status:** Fixed

Description

settleWithdrawRequest() loops through **withdrawalRequests** and transfers tokens to users:

Vault.sol#L476-L491

```
// Transfer withdrawal amount to the user.
if (tokenBalance < withdrawalAmount) {
    // ...

    // Transfer remaining token balance to the user.
    IERC20(withdrawalRequests[i].tokenAddress).transfer(user, tokenBalance);
} else {
    // Transfer the full withdrawal amount to the user and burn the corresponding…
    IERC20(withdrawalRequests[i].tokenAddress).transfer(user, withdrawalAmount);
    _burn(address(this), lpAmount);
}
```

However, if the transfer of tokens to any user fails, the entire call to *settleWithdrawRequest()* will revert. For example:

- Alice deposits USDC into the vault.
- Alice gets blacklisted by Circle; her address can no longer receive USDC.
- Alice calls *addWithdrawRequest()* to withdraw USDC to her address.
- When *settleWithdrawRequest()* is called, the function reverts when attempting to transfer USDC to Alice.

Since there is no way to skip withdrawal requests in *settleWithdrawRequest()*, the function will always revert when called, making it impossible for users to withdraw funds.

Recommended mitigation

In *settleWithdrawRequest()*, consider implementing a way to skip individual requests.

Team response

Fixed by removing the withdrawal queue. Withdrawals are now executed instantly.

Mitigation review

Verified, the bug is fixed as there is no withdrawal queue.

TRST-H-2: Looping over withdrawal requests could lead to DOS

- **Category:** DOS attacks
- **Source:** Vault.sol
- **Status:** Fixed

Description

Users call *addWithdrawRequest()* to add their withdrawal request to either the **withdrawalRequests** or **pendingWithdrawalRequests** array. When the master address calls *settleWithdrawRequest()*, the function loops over both arrays:

Vault.sol#L463-L495

```
// Iterate through each withdrawal request.
for (uint256 i = 0; i < withdrawalRequests.length; ++i) {
    address user = withdrawalRequests[i].userAddress;
    uint256 lpAmount = withdrawalRequests[i].lpAmount;

    // Readacted code that processes the withdrawal
}
```

Vault.sol#L517-L525

```
// Move pending withdrawal requests to the confirmed withdrawal requests array.
for (uint256 i = 0; i < pendingWithdrawalRequests.length; i++) {
    withdrawalRequests.push(pendingWithdrawalRequests[i]);
}
```

However, this is dangerous as both arrays are unbounded – users can call *addWithdrawRequest()* repeatedly to keep adding withdrawal requests to either array.

If the array grows too large, the amount of gas needed to loop over and process each individual request might exceed the block gas limit, causing *settleWithdrawRequest()* to always revert when called. This will make it impossible for users to withdraw funds.

Recommended mitigation

Instead of using arrays to implement withdrawal queues, consider using a mapping with three indexes instead:

```
uint256 start;
uint256 mid;
uint256 end;
mapping(uint256 => WithdrawalInfo) withdrawalRequests;
```

To add new withdrawal requests to the queue in *addWithdrawRequest()*, store the new request at the **end** index and increment it by one:

```
withdrawalRequests[end++] = newWithdrawal;
```

When *activatePendingStatus()* is called, store the current **end** index in **mid**:

```
mid = end;
```

mid represents the index that *settleWithdrawRequest()* should iterate up to – all requests from **start** to **mid** are in the withdrawal queue, and the remaining requests from **mid** to **end** are in the pending withdrawal queue.

In *settleWithdrawRequest()*, remove requests from the front of the queue by incrementing the **start** index:

```
// Iterate through each withdrawal request.
for (uint256 i = start; i < mid; ++i) {
    WithdrawalInfo memory withdrawal = withdrawalRequests[i];
    // ...
}
start = mid;
```

This approach achieves the same functionality as using two arrays, but the code only manages one queue, which removes the need to copy the entire **pendingWithdrawalRequests** array into **withdrawalRequests**.

To resolve the problem of DOS in *settleWithdrawRequest()*, add a parameter to specify the number of requests to loop through:

```
function settleWithdrawRequest(uint256 noOfRequests) ... {
    uint256 queueLength = mid - start;
    noOfRequests = noOfRequests < queueLength ? noOfRequests : queueLength
    for (uint256 i = 0; i < noOfRequests; i++) {
        WithdrawalInfo memory withdrawal = withdrawalRequests[start + i];
        // ...
    }
    start += noOfRequests;
}
```

If the queue ever grows too large to be processed in one function call, the master address can simply call *settleWithdrawRequest()* multiple times instead.

Team response

Fixed by removing the withdrawal queue. Withdrawals are now executed instantly.

Mitigation review

Verified, the bug is fixed as there is no withdrawal queue.

TRST-H-3: *createGMOrder()* shouldn't transfer tokens for decrease orders

- **Category:** Logical flaws
- **Source:** GmxPlugin.sol
- **Status:** Fixed

Description

createGMOrder() sends **initialCollateralToken** to GMX's order vault for all order types:

GmxPlugin.sol#L584-L591

```
// Send execution fee to orderVault
_exchangeRouter.sendWnt{value: executionFee}(routerConfig.orderVault, executionFee);

// Transfer initialCollateralToken to orderVault
_exchangeRouter.sendTokens(initialCollateralToken, routerConfig.orderVault,
    initialCollateralDeltaAmount);

// Create the order using the external exchange router
_exchangeRouter.createOrder(_params);
```

However, collateral only needs to be transferred for swap/increase position orders. According to GMX's documentation, **initialCollateralDeltaAmount** is the amount of tokens to withdraw for **MarketDecrease**, **LimitDecrease** and **StopLossDecrease** orders.

As such, if *createGMOrder()* is ever used to execute a decrease order, tokens will be wrongly transferred to GMX's order vault, causing a loss of funds for users and the protocol.

Recommended mitigation

Check if the order is a swap/increase order before sending tokens to GMX's order vault:

```
if (
    _params.orderType == IExchangeRouter.OrderType.MarketSwap ||
    _params.orderType == IExchangeRouter.OrderType.LimitSwap ||
    _params.orderType == IExchangeRouter.OrderType.MarketIncrease ||
    _params.orderType == IExchangeRouter.OrderType.LimitIncrease
) {
    // Approve initialCollateralToken transfer
    IERC20(initialCollateralToken).approve(routerConfig.router,
        initialCollateralDeltaAmount);

    // Transfer initialCollateralToken to orderVault
    _exchangeRouter.sendTokens(initialCollateralToken, routerConfig.orderVault,
        initialCollateralDeltaAmount);
}
```

Team response

Fixed as recommended.

Mitigation review

Verified, *createGMOrder()* only transfers **initialCollateralToken** to GMX's order vault for swap and increase orders.

TRST-H-4: `pnlFactorType` is encoded wrongly in *getMarketTokenPrice()*

- **Category:** Logical flaws
- **Source:** GmxPlugin.sol
- **Status:** Fixed

Description

When calling the *getMarketTokenPrice()* function of GMX's reader, **pnlFactorType** is encoded with *abi.encodePacked()*:

GmxPlugin.sol#L621

```
bytes32 pnlFactorType = keccak256(abi.encodePacked("MAX_PNL_FACTOR_FOR_TRADERS"));
```

However, it should be encoded with *abi.encode()* instead according to GMX's code:

Keys.sol#L263-L264

```
// @dev key for max pnl factor
bytes32 public constant MAX_PNL_FACTOR_FOR_TRADERS =
    keccak256(abi.encode("MAX_PNL_FACTOR_FOR_TRADERS"));
```

Due to an invalid **pnlFactorType**, get*MarketTokenPrice()* will return an incorrect price for GM tokens. More specifically, the price returned would not factor in the PnL of the market, causing all accounting in the plugin to be incorrect.

Recommended mitigation

Modify the code to use *abi.encode()* instead:

```
- bytes32 pnlFactorType = keccak256(abi.encodePacked("MAX_PNL_FACTOR_FOR_TRADERS"));
+ bytes32 pnlFactorType = keccak256(abi.encode("MAX_PNL_FACTOR_FOR_TRADERS"));
```

Team response

Fixed as recommended.

Mitigation review

Verified, **pnlFactorType** is now encoded correctly.

TRST-H-5: *totalAssetInUsd()* will be decreased while a GMX request is pending

- **Category:** Logical flaws
- **Source:** GmxPlugin.sol
- **Status:** Fixed

Description

When creating a GMX deposit, withdrawal or order request, long/short/market tokens are sent to their respective vaults and temporarily held there. Afterwards, GMX's keepers that listen for request transactions will send a separate transaction to execute the request.

However, the issue is that *GMXPlugin.getTotalLiquidity()* and *Vault.totalAssetInUsd()* use the current balance of tokens in their respective contracts to calculate the total value of assets in the vault.

As such, during the period when a request is sent but hasn't been executed by GMX's keepers, both functions will temporarily return values much smaller than expected.

For example:

- Transaction 1: Call *sellGMToken()* to withdraw assets from the GM pool
  - This transfers GM tokens to GMX's withdrawal vault.
  - A withdrawal request is created in GMX.
- Transaction 2: A user calls *addDepositRequest()*
  - The GM tokens sent in transaction 1 are still in GMX's withdrawal vault.
  - Therefore, the total value returned by *totalAssetInUsd()* is much smaller as it excludes the value of GM tokens that are pending withdrawal.
- Transaction 3: GMX's keeper executes the withdrawal request.
  - This transfers long and short tokens to the Vault contract.

This is problematic as *addDepositRequest()* can be called by users at any time. If users manage to deposit their assets between transaction 1 and transaction 3 as described above, they will unfairly receive more shares as the total value of the vault is temporarily decreased.

Recommended mitigation

Consider preventing users from depositing when there is a pending GMX request. One way to achieve this would be to disable *addDepositRequest()* when *execute()* is called, and re-enable it when a callback in **gmxParams.callbackContract** is called.

Team response

Fixed by disabling deposits, withdrawals, and orders whenever there is a pending GMX request. This is done using GMX's callbacks in GmxCallback.sol.

Mitigation review

GmxCallback.sol resumes deposits, withdrawals, and orders whenever a pending order is frozen by removing its key:

GmxCallback.sol#L186-L188

```
function afterOrderFrozen(bytes32 key, ...) external onlyHandler(State.Order) {
    removeKey(key, State.Order);
}
```

This is incorrect – the key should not be removed when an order is frozen.

Users should not be able to deposit/withdraw whenever an order is still pending as a portion of the protocol's assets are still held in GMX's order vault. Functionality should only resume after the frozen order is cancelled as the frozen assets will be returned to the GmxPlugin contract.

Additionally, cancelling the frozen order will call *removeKey()* again, so the same key will be removed twice.

Team response

Fixed by not removing the order key in the *afterOrderFrozen()* callback.

Mitigation review

Verified, deposits and withdrawals will not resume when an order is frozen. They will only be resumed after the order is cancelled.

Medium severity findings

TRST-M-1: *convertAssetToLP()* returns 0 if the vault has assets before the first deposit

- **Category:** Logical flaws
- **Source:** Vault.sol
- **Status:** Fixed

Description

convertAssetToLP() is used to determine how much vault tokens to mint to users when they deposit funds:

Vault.sol#L809-L819

```
// Convert asset amount to LP tokens based on current total asset and LP token supply.
function convertAssetToLP(uint256 _amount) public view returns (uint256) {
    // If the total asset is zero, perform direct decimal conversion.
    uint256 _totalAssetInUsd = totalAssetInUsd() > protocolFeeInVault ?
        totalAssetInUsd() - protocolFeeInVault : 0;
    if (_totalAssetInUsd == 0) {
        return convertDecimals(_amount, ASSET_DECIMALS, MOZAIC_DECIMALS);
    }

    // Perform conversion based on proportion of the provided amount to total asset.
    return (_amount * totalSupply()) / _totalAssetInUsd;
}
```

The **_totalAssetInUsd == 0** check exists as *totalSupply()* and **_totalAssetInUsd** are 0 on the first deposit, which makes calculation based on proportions not possible.

However, an attacker can skip this check by donating assets to the vault (e.g. transfer 1 wei of any accepted token directly to the contract), which would make **_totalAssetInUsd** non-zero.

If this occurs before the first deposit, since *totalSupply()* is still 0, *convertAssetToLP()* will always return 0 for all future deposits. This will make it impossible for anyone to deposit into the vault as *addDepositRequest()* reverts when *convertAssetToLP()* returns 0.

Recommended mitigation

Modify the check to also ensure that *totalSupply()* is non-zero:

```
-   if (_totalAssetInUsd == 0) {
+   if (_totalAssetInUsd == 0 || totalSupply() == 0) {
        return convertDecimals(_amount, ASSET_DECIMALS, MOZAIC_DECIMALS);
    }
```

Team response

Fixed as recommended.

Mitigation review

Verified, *convertAssetToLP()* now converts assets to shares at a 1:1 ratio if *totalSupply()* is zero, preventing the exploit described above.

TRST-M-2: Calculation in *convertAssetToLP()* is susceptible to inflation attacks

- **Category:** Logical flaws
- **Source:** Vault.sol
- **Status:** Fixed

Description

convertAssetToLP() is used to determine how much vault tokens to mint to users when they deposit funds:

Vault.sol#L809-L819

```
// Convert asset amount to LP tokens based on current total asset and LP token supply.
function convertAssetToLP(uint256 _amount) public view returns (uint256) {
    // If the total asset is zero, perform direct decimal conversion.
    uint256 _totalAssetInUsd = totalAssetInUsd() > protocolFeeInVault ?
        totalAssetInUsd() - protocolFeeInVault : 0;
    if (_totalAssetInUsd == 0) {
        return convertDecimals(_amount, ASSET_DECIMALS, MOZAIC_DECIMALS);
    }

    // Perform conversion based on proportion of the provided amount to total asset.
    return (_amount * totalSupply()) / _totalAssetInUsd;
}
```

This is vulnerable to inflation attacks, where an attacker donates assets directly to the vault to inflate **_totalAssetInUsd** while *totalSupply()* is small. For example:

- Alice, the first depositor, deposits tokens worth only 0.000001 USD.
- In *convertAssetToLP()*, **_amount** is 1e30, thus only 1 share is minted to Alice.
- Alice transfers 1000 USD worth of tokens to the vault to inflate **_totalAssetInUsd**.
- Bob deposits 500 USD worth of tokens into the vault.
- Since *totalSupply()* is 1 and **_amount** is less than **_totalAssetInUsd**, *convertAssetToLP()* rounds down to 0, causing *addDepositRequest()* to revert.

A first depositor can abuse this to prevent other users from depositing into the vault.

Recommended mitigation

On the first deposit, mint a small amount of vault tokens to a dead address:

```
    if (_totalAssetInUsd == 0) {
+       _mint(address(0), 10 ** MOZAIC_DECIMALS);
        return convertDecimals(_amount, ASSET_DECIMALS, MOZAIC_DECIMALS);
    }
```

Team response

Fixed by adding a minimum deposit check in *addDepositRequest()* at Vault.sol#L357.

Mitigation review

The current fix does not work – an attacker can bypass the check by depositing the minimum amount and withdrawing all LP tokens except 1 to make *totalSupply()* extremely small. This is possible as *addWithdrawalRequest()* now processes withdrawals instantly, instead of having a withdrawal queue.

Consider implementing the recommended fix above.

Team response

Fixed by adding virtual shares and assets in *totalSupply()* and *totalAssetInUsd()* respectively.

Mitigation review

Verified, *totalSupply()* will never go below 1e6 so the inflation attack described above is no longer possible.

TRST-M-3: *getCurrentLiquidityProviderRate()* returns 0 when _totalAssets is small

- **Category:** Rounding issues
- **Source:** Vault.sol
- **Status:** Fixed

Description

getCurrentLiquidityProviderRate() calculates **lpRate**, which is the USD value of each vault token (note that **ASSET_DECIMALS** is 36 and **MOZAIC_DECIMALS** is 6):

Vault.sol#L670-L674

```
// Convert total assets to the desired decimals
uint256 adjustedAssets = convertDecimals(_totalAssets, ASSET_DECIMALS,
    MOZAIC_DECIMALS);

// Calculate the current rate
currentRate = adjustedAssets * 1e18 / totalSupply();
```

To convert **_totalAssets** from 36 to 6 decimals, **_totalAssets** is divided by 1e30. However, if **_totalAssets** is smaller than 1e30, **adjustedAssets** will round down to 0, causing *getCurrentLiquidityProviderRate()* to return 0 as the new rate.

This is problematic in *upateLiquidityProviderRate()* as it will update **lpRate** to 0, causing deposits to wrongly accrue to the protocol fee:

- Assume the first depositor deposits less than 0.000001 USD worth of tokens.
- The master address calls *upateLiquidityProviderRate()*, which updates **lpRate** to 0 as mentioned above since **_totalAssets** is less than 1e30.
- Another user deposits some tokens.

- When *upateLiquidityProviderRate()* is called afterwards, the new **lpRate** will be 1e18. This causes part of the user's deposit to accrue to the protocol fee, even though the vault hasn't generated any profit.

Recommended mitigation

In *getCurrentLiquidityProviderRate()*, check that **_totalAssets** is sufficiently large:

Vault.sol#L666-L669

```
   // Check if total supply or total assets is zero
- if (totalSupply() == 0 || _totalAssets == 0) {
+ if (totalSupply() == 0 || _totalAssets < 10 ** (ASSET_DECIMALS - MOZAIC_DECIMALS)) {
      currentRate = 1e18;
  } else {
```

Team response

Fixed as recommended.

Mitigation review

Verified, *getCurrentLiquidityProviderRate()* now returns **currentRate** as 1e18 when **_totalAssets** is small, preventing the exploit described above.

TRST-M-4: Unsafe cast of GMX market token price

- **Category:** Logical flaws
- **Source:** GmxPlugin.sol
- **Status:** Fixed

Description

In *getTotalLiquidity()*, the GMX market token price returned by *getMarketTokenPrice()* is cast to uint256 directly:

GmxPlugin.sol#L289

```
uint256 marketTokenPrice = uint256(getMarketTokenPrice(pools[i].poolId, true));
```

According to GMX's documentation, although unlikely, it is possible for a market's token to have a negative price:

> *It is rare but possible for a pool's value to become negative, this can happen since the impactPoolAmount and pending PnL is subtracted from the worth of the tokens in the pool*

As such, casting directly to uint256 is dangerous as **marketTokenPrice** will become an extremely large value when *getMarketTokenPrice()* returns a negative price. Should this occur, it would inflate the plugin's total liquidity by a huge amount, causing all accounting in the protocol to malfunction.

Recommended mitigation

If a market's token price happens to be negative, consider skipping this market and not adding its value to the plugin's total liquidity:

```
int256 marketTokenPrice = getMarketTokenPrice(pools[i].poolId, true)
if (marketTokenPrice <= 0) continue;
uint256 amount = marketTokenBalance * uint256(marketTokenPrice);
```

Team response

Fixed as recommended.

Mitigation review

Due to changes in Vault.sol, the recommended fix is no longer comprehensive.

getPoolTokenPrice() in Vault.sol calls *getMarketTokenPrice()*, which is now named *getPoolTokenPrice()*, and performs an unsafe cast:

Vault.sol#L432

```
return uint256(IPlugin(plugin).getPoolTokenPrice(_poolId, true));
```

As *getPoolTokenPrice()* is used in *addWithdrawalRequest()*, if the GM token happens to have a negative price, users will end up creating withdrawal requests with 0 GM tokens. This will revert in GMX's execution, and the request will be cancelled.

To prevent users from creating unfulfillable withdrawal requests, consider reverting when the GM token price is negative:

Vault.sol#L432

```
int256 price = IPlugin(plugin).getPoolTokenPrice(_poolId, true));
if (price < 0) revert("Vault: GM token price is negative.");
return uint256 price;
```

Team response

Fixed by adding a require statement that ensures **price** is greater than 0.

Mitigation review

Verified, *getPoolTokenPrice()* will revert if the GM token's price is negative or 0.

TRST-M-5: Missing slippage checks for GMX deposits and withdrawals

- **Category:** Logical flaws
- **Source:** GmxPlugin.sol
- **Status:** Fixed

Description

minMarketToken, which is the minimum output amount of GM token, is specified as 0 in *buyGMToken()*:

GmxPlugin.sol#L479-L492

```
IExchangeRouter.CreateDepositParams memory params =
IExchangeRouter.CreateDepositParams(
    address(this),                    // receiver
    gmxParams.callbackContract,       // callbackContract
    gmxParams.uiFeeReceiver,          // uiFeeReceiver
    marketAddress,
    longToken,
    shortToken,
    longTokenSwapPath,
    shortTokenSwapPath,
    0,                                // minMarketTokens
    gmxParams.shouldUnwrapNativeToken, // shouldUnwrapNativeToken
    executionFee,
    gmxParams.callbackGasLimit        // callbackGasLimit
);
```

Likewise, in *sellGMToken()*, **minLongTokens** and **minShortTokens**, which is the minimum output amount of long and short tokens, is also 0:

GmxPlugin.sol#L537-L549

```
IExchangeRouter.CreateWithdrawalParams memory params =
IExchangeRouter.CreateWithdrawalParams(
    localVault,                       // receiver
    gmxParams.callbackContract,       // callbackContract
    gmxParams.uiFeeReceiver,          // uiFeeReceiver
    marketAddress,
    longTokenSwapPath,
    shortTokenSwapPath,
    0,                                // minLongTokens
    0,                                // minShortTokens
    gmxParams.shouldUnwrapNativeToken, // shouldUnwrapNativeToken
    executionFee,
    gmxParams.callbackGasLimit        // callbackGasLimit
);
```

This leaves deposits and withdrawals vulnerable to slippage, especially since deposit/withdrawal requests are not executed instantly, but reliant on GMX's keepers to execute the request.

The amount of GM token or short/long token received after the request is executed could be much smaller than the expected amount when the request was sent by the plugin, potentially causing a loss of funds.

Recommended mitigation

Consider not leaving **minMarketTokens**, **minLongTokens** and **minShortTokens** as 0 in deposit/withdrawal requests.

Either add parameters to *buyGMToken()* and *sellGMToken()* for the caller to specify their own slippage values, or calculate their values based on the current state of the GMX market.

Team response

Fixed by adding slippage parameters in *buyGMToken()* and *sellGMToken()*. These slippage parameters can be specified by users when they deposit/withdraw.

Mitigation review

addDepositRequest() in Vault.sol is missing the slippage parameter. This can be seen from the encoding of **payload** in *stakeToSelectedPool()*:

Vault.sol#L391-L392

```
// Encode the payload for the 'Stake' action using the selected plugin and pool.
bytes memory payload = abi.encode(uint8(selectedPoolId), allowedTokens, _amounts);
```

The **minGmAmount** parameter, which is supposed to be the fourth argument, is missing.

Team Response

Fixed by adding the missing **minGmAmount** parameter to *stakeToSelectedPool()*.

Mitigation review

Verified, it is now possible to specify slippage for GMX deposits when calling *addDepositRequest()*.

TRST-M-6: *getMarketTokenPrice()* will always revert for some markets

- **Category:** Logical flaws
- **Source:** GmxPlugin.sol
- **Status:** Acknowledged

Description

getMarketTokenPrice() uses *getTokenPriceInfo()* to fetch the prices of tokens, including a market's index token:

GmxPlugin.sol#L615-L616

```
// Fetch token prices for indexToken, longToken, and shortToken
IPrice.Props memory indexTokenPrice = getTokenPriceInfo(_pool.indexToken);
```

getTokenPriceInfo() attempts to retrieve the index token's decimals as such:

GmxPlugin.sol#L644

```
uint256 tokenDecimal = IERC20Metadata(token).decimals();
```

However, for some markets, the address stored at **_pool.indexToken** is not a contract. These markets are:

- Synthetic markets:
  - BTC / USD
  - DOGE / USD
  - LTC / USD
  - XRP / USD
- Stablecoin markets, where **_pool.indexToken** is the zero address:
  - USDC / USDC.e
  - USDC / USDT
  - USDC / DAI

For these markets, attempting to fetch the decimals of the index token as shown above will revert. As such, the current implementation of GMXPlugin.sol is incompatible with these markets since *getMarketTokenPrice()* will always revert.

Recommended mitigation

The decimals for all index tokens can be found here. For synthetic markets, consider storing these decimals in a mapping and using it in *getTokenPriceInfo()*, instead of calling *IERC20Metadata.decimals()*.

For stablecoin markets where index token is the zero address, set **indexTokenPrice** to 0. This is what GMX does internally.

Team response

Acknowledged.

TRST-M-7: GMXPlugin.sol lacks functionality to handle frozen orders

- **Category:** Logical flaws
- **Source:** GmxPlugin.sol
- **Status:** Fixed

Description

According to GMX's documentation, keepers can choose to freeze orders during execution if:

- The price impact is too high, and the acceptable price is not met.
- There is insufficient liquidity to fulfil the order.

Frozen orders will only be executed by GMX's keepers once the acceptable price can be met, which might take an extremely long period of time. While the order is frozen, funds for the order will be held in GMX.

Orders can be unfrozen in two ways:

- Calling *ExchangeRouter.updateOrder()* to update the order's parameters, as described here.
- Calling *ExchangeRouter.cancelOrder()* to cancel the order.

However, GMXPlugin.sol does not have functionality to call either of the two functions.

Therefore, if an order from the plugin is ever frozen, the assets for that order might be stuck in GMX for an extended duration, and possibly stuck forever if the order never becomes executable.

Recommended mitigation

Consider adding the ability to update or cancel an order in *execute()*.

Team response

Fixed by giving the master address the ability to cancel orders.

Mitigation review

Verified, frozen orders can now be cancelled with *cancelOrder()* in GmxPlugin.sol.

TRST-M-8: Assets from cancelled GMX requests are not handled

- **Category:** Logical flaws
- **Source:** GmxPlugin.sol
- **Status:** Fixed

Description

When a pending GMX request is cancelled – either by GMX or calling *cancelAction()*, assets for that request are refunded to the address that created the request. For this protocol, it would be the GmxPlugin contract as it calls *createDeposit(), createWithdrawal()* and *createOrder()*.

When *stake()* or *createOrder()* is called, it pulls assets from the Vault contract and transfers them to GMX.

However, when the deposit/order request is cancelled, assets that are refunded from GMX are not sent back to the Vault contract. Calling *stake()* or *createOrder()* again would also not work as it will pull in new assets from the vault.

As such, the refunded assets will be stuck in the GmxPlugin contract.

Recommended mitigation

When deposit or order requests are cancelled, consider sending the refunded assets back to the vault. This can be done using the *afterDepositCancellation()* and *afterOrderCancellation()* callbacks in GmxCallback.sol.

Team response

Fixed by adding a *transferAllTokensToVault()* function that transfers long and short tokens from the GmxPlugin contract to the vault, and calling it all callbacks.

Mitigation review

Verified, refunded assets from request cancellations will now be sent back to the vault in their respective callbacks.

TRST-M-9: TokenPriceConsumer.sol does not validate Chainlink feeds

- **Category:** Logical flaws
- **Source:** TokenPriceConsumer.sol
- **Status:** Fixed

Description

TokenPriceConsumer.getTokenPrice() uses Chainlink's *latestRoundData()* to fetch the price of an asset:

TokenPriceConsumer.sol#L33-L35

```
(, int256 answer, , , ) = priceFeed.latestRoundData();
// Token price might need additional scaling based on decimals
return uint256(answer);
```

However, it does not have any checks to ensure that the price returned is updated or acceptable. Chainlink's price feed might return incorrect prices if:

- The price feed is stale.
- Arbitrum's sequencer is down, so prices cannot be updated.

Recommended mitigation

Consider adding the relevant checks to *getTokenPrice()*:

- Chainlink's documentation on the sequencer uptime check
- GMX's implementation

For example:

```
function getTokenPrice(address tokenAddress) public view returns (uint256) {
    AggregatorV3Interface priceFeed = tokenPriceFeeds[tokenAddress];
    require(address(priceFeed) != address(0), "Price feed not found");

    // Check if the sequencer is up
    (int256 answer, uint256 startedAt , , ) = sequencerUptimeFeed.latestRoundData();
    require(
        answer == 0 && block.timestamp - startedAt >= GRACE_PERIOD_TIME,
        "PriceFeed: Sequencer is down"
    );

    (
        uint80 roundId,
        int256 answer, ,
        uint256 updatedAt,
    ) = priceFeed.latestRoundData();

    // Sanity check
    require(roundId != 0 && answer >= 0 && updatedAt != 0, "PriceFeed: Sanity check");

    // Stale price check
    uint256 heartbeatDuration = tokenHeartbeatDurations[tokenAddress];
    require(block.timestamp - updatedAt <= heartbeatDuration, "Price feed is stale");

    // Token price might need additional scaling based on decimals
    return uint256(answer);
}
```

The heartbeat duration of each price feed can be checked in their respective pages by hovering over "Trigger parameters". For example, BTC / USD has a heartbeat duration of 3600 seconds.

Team response

Fixed by adding the sanity and stale price check.

Mitigation review

Verified. Note that since the sequencer uptime check was not added, if Arbitrum's sequencer was to go down, the price returned by Chainlink's feeds would be stale but users can still interact with the vault through L1 -> L2 transactions.

Low severity findings

TRST-L-1: Unsafe ERC-20 transfers or approvals

- **Category:** ERC20 compatibility issues
- **Source:** Vault.sol, GmxPlugin.sol
- **Status:** Fixed

Description

ERC-20's *transfer()*, *transferFrom()* and *approve()* are used throughout the code:

- *transfer()*:
  - Vault.sol#L486
  - Vault.sol#L489
- *transferFrom()*:
  - GmxPlugin.sol#L576
- *approve()*:
  - Vault.sol#L552
  - Vault.sol#L578
  - GmxPlugin.sol#L496
  - GmxPlugin.sol#L500
  - GmxPlugin.sol#L552
  - GmxPlugin.sol#L579

These functions will not work for tokens that are not ERC-20 compliant. This currently is not a problem as there is no GMX market with a non-compliant token, but could become a problem in the future.

Recommended mitigation

Use *safeTransfer()*, *safeTransferFrom()* and *safeIncreaseAllowance()* from OpenZeppelin's SafeERC20 instead.

Team response

Fixed as recommended.

Mitigation review

Verified, there are no more unsafe ERC-20 approvals/transfers.

TRST-L-2: *stakeToSelectedPool()* doesn't handle duplicate tokens in `allowedTokens`

- **Category:** Logical flaws
- **Source:** Vault.sol
- **Status:** Fixed

Description

stakeToSelectedPool() loops over **allowedTokens** and calls *execute()* to stake tokens:

Vault.sol#L396-L408

```
for (uint256 i = 0; i < allowedTokens.length; i++) {
    if (allowedTokens[i] == _token) {
        // ...

        // Execute the 'Stake' action on the selected plugin with the encoded payload.
        this.execute(uint8(selectedPluginId), IPlugin.ActionType.Stake, payload);
    }
}
```

Since the function does not return after calling *execute()*, if **allowedTokens** contains duplicate tokens, the function will wrongly stake twice. Additionally, if **allowedTokens** does not contain **_token**, *stakeToSelectedPool()* also does not revert.

Recommended mitigation

Modify *stakeToSelectedPool()* to handle the cases mentioned above:

```
  function stakeToSelectedPool(address _token, uint256 _tokenAmount) internal {
      // Redacted code...

      // Iterate through the allowed tokens to find the matching token.
      for (uint256 i = 0; i < allowedTokens.length; i++) {
          if (allowedTokens[i] == _token) {
              // Redacted code...
+             return;
          }
      }
+     revert("Vault: deposit token not in allowedTokens");
  }
```

Team response

Fixed by adding the recommended return statement.

Mitigation review

Verified, *stakeToSelectedPool()* will only stake for the first occurrence of **_token** in **_allowedTokens**.

Note that the recommended revert statement was not added, as such, if **_token** is not in **_allowedTokens**, users can deposit without staking into any plugin.

Additional recommendations

TRST-R-1: Use non-upgradeable ReentrancyGuard

Both Vault and GMXPlugin contracts inherit ReentrancyGuardUpgradeable. However, since both contracts are not meant to be upgradeable, consider using the non-upgradeable version of OpenZeppelin's ReentrancyGuard instead.

TRST-R-2: Unnecessary "this." in *stakeToSelectedPool()*

Since *execute()* is declared public, there is no need to call *this.execute()*:

Vault.sol#L406

```
- this.execute(uint8(selectedPluginId), IPlugin.ActionType.Stake, payload);
+ execute(uint8(selectedPluginId), IPlugin.ActionType.Stake, payload);
```

TRST-R-3: Deleting `withdrawalRequests[i]` is redundant in *settleWithdrawRequest()*

Since the entire **withdrawalRequests** array is cleared later in the function, deleting individual elements in the for-loop is redundant. The following lines can be removed:

Vault.sol#L492-L494

```
- // Clear the processed withdrawal request.
- delete withdrawalRequests[i];
```

TRST-R-4: Code in *settleWithdrawRequest()* can be simplified

Use the delete keyword to clear the **withdrawalRequests** array instead:

Vault.sol#L497-L500

```
  // Clear the entire withdrawal requests array.
- while (withdrawalRequests.length > 0) {
-     withdrawalRequests.pop();
- }
+ delete withdrawRequests;
```

TRST-R-5: Typo in upateLiquidityProviderRate()

"upate" should be "update":

Vault.sol#L583

```
- function upateLiquidityProviderRate() external onlyMaster nonReentrant {
+ function updateLiquidityProviderRate() external onlyMaster nonReentrant {
```

TRST-R-6: Only 256 plugins can be used

In Vault.sol, *addPlugin()* does not have a maximum number of plugins that can be added.

However, *totalAssetInUsd()* iterates over **plugins** with a uint8 index. This implicitly limits the maximum number of plugins that can be added to 256, since *totalAssetInUsd()* would revert if there are more plugins.

To remove this implicit limit, consider using uint256 for the index instead:

Vault.sol#L683

```
- for (uint8 i; i < plugins.length; ++i) {
+ for (uint256 i; i < plugins.length; ++i) {
```

TRST-R-7: Master address is not used in GMXPlugin.sol

In GMXPlugin.sol, the *onlyMaster()* modifier is not used anywhere in the code, which makes storing the master address redundant as it cannot call any function in the contract.

Consider removing all master-related functionality, such as *onlyMaster()*, *setMaster()* and the **master** state variable from the contract altogether.

TRST-R-8: *convertDecimals()* can be used to simplify the code

In *calculateTokenValueInUsd()*:

Vault.sol#L770-L777, GmxPlugin.sol#L319-L328

```
  // Adjust the token amount based on the difference in decimals.
- if (tokenDecimals + priceConsumerDecimals >= ASSET_DECIMALS) {
-     decimalsDiff = tokenDecimals + priceConsumerDecimals - ASSET_DECIMALS;
-     return (_tokenAmount * tokenPrice) / (10 ** decimalsDiff);
- } else {
-     decimalsDiff = ASSET_DECIMALS - tokenDecimals - priceConsumerDecimals;
-     return (_tokenAmount * tokenPrice * (10 ** decimalsDiff));
- }
+ return convertDecimals(
+     _tokenAmount * tokenPrice,
+     tokenDecimals + priceConsumerDecimals,
+     ASSET_DECIMALS
+ );
```

In *getTotalLiquidity()*:

GmxPlugin.sol#L292-L307

```
  // Use IERC20Metadata only once to get decimals.
  uint256 decimals = IERC20Metadata(marketTokenAddress).decimals()
      + MARKET_TOKEN_PRICE_DECIMALS;

- // Refactor decimalsDiff calculation to improve readability.
- uint256 decimalsDiff = abs(int256(decimals) - int256(ASSET_DECIMALS));
- uint256 adjustedAmount;
-
- // Adjust amount based on decimalsDiff.
- if (decimals >= ASSET_DECIMALS) {
-     adjustedAmount = amount / 10**decimalsDiff;
- } else {
-     adjustedAmount = amount * 10**decimalsDiff;
- }
-
- // Accumulate adjustedAmount to totalAsset.
- totalAsset += adjustedAmount;
+ totalAsset += convertDecimals(amount, decimals, ASSET_DECIMALS);
```

Centralization risks

TRST-CR-1: Thesus Vault risks

Vault.sol, and by extension, GMXPlugin.sol, should be considered fully centralized.

The owner address can:

- Set **tokenPriceConsumer**, which is a contract that returns prices for tokens. This allows a malicious owner to manipulate the prices of tokens in the vault.
- Add/remove tokens from **acceptedTokens.** By removing tokens from **acceptedTokens**, the owner can control the value of the vault and use this to drain the vault's balance.
- Add/remove plugins. The owner can add a malicious plugin and approve it to transfer all tokens in the vault out using *approveTokens()*.

The master address is responsible for:

- Managing the vault's funds through GMX, which could result in a loss of funds if they are not managed properly.