SECURITY REVIEW LEXER MARKETS

Summary

Auditors: 0xWeiss (Marc Weiss) and 0xKato

Client: Lexer Markets

Report Delivered: 14 November, 2023

Protocol Summary

Protocol Name	Lexer Markets
Language	Solidity
Codebase	https://github.com/lexer-markets/contracts-audit/tree/6894013e601be458c57f5bde6f88290f99588225
Commit	6894013e601be458c57f5bde6f88290f99588225
Previous Audits	None
Test Coverage	Poor ✗
Key Management	Good ✓
Centralization	Very Centralized

About 0xWeiss and 0xKato

0xWeiss and 0xKato are independent security researchers. Having found numerous security vulnerabilities in various DeFi protocols, they do their best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Reach out on Twitter @0xWeisss @0xKato.

Audit Summary

Lexer Markets engaged 0xWeiss and 0xKato through Hyacinth to review the security of its DeFi protocol. From the 19th of September to the 25th of October, 0xWeiss and 0xKato reviewed the source code in scope.

Lexer is a highly permissioned protocol with several roles and keepers. They have control over key state changes on the smart contracts.

There is always a risk for the protocol to have their keys compromised, fortunately, Lexer uses a set of timelocks and a good internal key manadgement pipeline.

The role manadgement has been structured in a way that there is not a single point of failure, including the usage of multi-signature and hardware wallets.

Lexer was forked from GMX V1 and modified to integrate with Synthetic assets. We spent over 10 auditor weeks reviewing the codebase, however given the number of findings discovered by us, having a clear security pipeline is key in the lifetime of the DeFi protocol, and we are always going to recommend having extra security reviews, just in case.

At the end, there were 49 issues identified. All findings have been recorded in the following report. For a detailed understanding of risk severity and potential attack vectors, refer to the complete audit report below.

A big percentage of the issues are in both the Synthetic version and the non-synthetic version of the contracts, it is probable that we missed pointing out that a specific issue is in different contracts because we did not want to raise the same issue twice. Consider checking whether issues are in both modules of the codebase or just in one.

Vulnerability Summary

Severity	Total	Pending	Acknowledged	Par. resolved	Resolved
🔴 **HIGH**	**11**	**0**	**2**	**3**	**6**
🟡 **MEDIUM**	**23**	**0**	**12**	**4**	**7**
🟢 **LOW**	**11**	**0**	**7**	**0**	**4**
🔵 **INF**	**4**	**0**	**0**	**1**	**3**

AUDIT SCOPE

CORE

ID	File Path
OB	contracts/core/Orderbook.sol
PM	contracts/core/PositionManager.sol
PR	contracts/core/PositionRouter.sol
ROU	contracts/core/Router.sol
PU	contracts/core/PositionUtils.sol
BPM	contracts/core/BasePositionManager.sol
SBPM	contracts/core/SBasePositionManager.sol
SHORTS	contracts/core/ShortsTracker.sol
SPM	contracts/core/SPositionManager.sol
SPR	contracts/core/SPositionRouter.sol
SPU	contracts/core/SPositionUtils.sol
SVAU	contracts/core/SVault.sol
VAU	contracts/core/Vault.sol
SVU	contracts/core/SVaultUtils.sol
VU	contracts/core/VaultUtils.sol
VPF	contracts/core/VaultPriceFeed.sol
VERR	contracts/core/VaultErrorController.sol
XLPM	contracts/core/XlpManager.sol
SXLPM	contracts/core/SXlpManager.sol

LEX

ELEX	contracts/lex/EsLex.sol
LEX	contracts/lex/Lex.sol
LF	contracts/lex/LexFloor.sol
LL	contracts/lex/LexLou.sol
LM	contracts/lex/LexMigrator.sol
MH	contracts/lex/MigrationHandler.sol
XLP	contracts/lex/XLP.sol

ORACLE

FPE	contracts/oracle/FastPriceEvents.sol
FPF	contracts/oracle/FastPriceFeed.sol
PF	contracts/oracle/PriceFeed.sol
PPF	contracts/oracle/PythPriceFeed.sol

PERIPHERALS

BU	contracts/peripherals/BalanceUpdate.sol
BS	contracts/peripherals/BatchSender.sol
ELBS	contracts/peripherals/EsLExBatchSender.sol
LT	contracts/peripherals/LexTimelock.sol
OBR	contracts/peripherals/OrderbookReader.sol
PRR	contracts/peripherals/PositionRouterReader.sol
PFT	contracts/peripherals/PriceFeedTimelock.sol
PPFT	contracts/peripherals/PythPriceFeedTimelock.sol
READ	contracts/peripherals/Reader.sol
RR	contracts/peripherals/RewardReader.sol
STT	contracts/peripherals/ShortTrackerTimelock.sol
SLT	contracts/peripherals/SLexTimelock.sol
SR	contracts/peripherals/SReader.sol
ST	contracts/peripherals/STimelock.sol
TIME	contracts/peripherals/Timelock.sol
VR	contracts/peripherals/VaultReader.sol

STAKING

SXLPMG	contracts/staking/StakedXlpMigrator.sol

Severity Classification

Severity	Classification
🔴 **HIGH**	Exploitable, causing loss/manipulation of assets or data.
🟡 **MEDIUM**	Risk of future exploits that may or may not impact the smart contract execution.
🟢 **LOW**	Minor code errors that may or may not impact the smart contract execution.
🔵 **INF**	No impact issues. Code improvement

Methodology

The auditing process pays special attention to the following considerations:

● Testing the smart contracts against both common and uncommon attack vectors.

● Assessing the codebase to ensure compliance with current best practices and industry standards.

● Ensuring contract logic meets the specifications and intentions of the client.

● Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.

● Thorough line-by-line manual review of the entire codebase by industry experts.

Findings and Resolutions

ID	Category	Severity	Status
OB-1	Price Manipulation	🔴 **HIGH**	Partially Resolved
OB-2	Incorrect Operators	🔴 **HIGH**	Resolved
SXLPMG-1	Uncompounded Rewards	🔴 **HIGH**	Resolved
SPR-1	Incorrect Design	🔴 **HIGH**	Resolved
SPR-2	Stale Prices	🔴 **HIGH**	Partially Resolved
VAU-1	Stale Prices	🔴 **HIGH**	Acknowledged
SVAU-1	DOS	🔴 **HIGH**	Resolved
SVAU-2	Loss of funds	🔴 **HIGH**	Partially Resolved
MH-1	Incorrect Design	🔴 **HIGH**	Resolved
GLOBAL-1	Loss of funds	🔴 **HIGH**	Acknowledged
VAU-2	Late Liquidations	🔴 **HIGH**	Resolved
OB-3	Loss of funds	🟡 **MEDIUM**	Partially Resolved
OB-4	Griefing	🟡 **MEDIUM**	Resolved
OB-5	Griefing	🟡 **MEDIUM**	Acknowledged
PR-1	Incorrect Validation	🟡 **MEDIUM**	Partially Resolved
PR-2	Malicious Keeper	🟡 **MEDIUM**	Acknowledged
PR-3	Malicious Keeper	🟡 **MEDIUM**	Acknowledged
SPR-3	Gas bomb	🟡 **MEDIUM**	Resolved
SPR-4	DOS	🟡 **MEDIUM**	Partially Resolved
SPR-5	DOS	🟡 **MEDIUM**	Acknowledged
SPR-6	Incorrect Validation	🟡 **MEDIUM**	Resolved
SPR-7	Griefing	🟡 **MEDIUM**	Acknowledged
SVAU-3	Broken Invariant	🟡 **MEDIUM**	Resolved
SVAU-4	Accounting manipulation	🟡 **MEDIUM**	Resolved

SVAU-5	Incorrect Validation	🟡	**MEDIUM**	Resolved
SVAU-6	Incorrect Design	🟡	**MEDIUM**	Acknowledged
SVAU-7	Incorrect Validation	🟡	**MEDIUM**	Acknowledged
SXLPM-1	Incorrect Validation	🟡	**MEDIUM**	Acknowledged
VPF-1	Price Manipulation	🟡	**MEDIUM**	Partially Resolved
PF-1	Deprecated function	🟡	**MEDIUM**	Acknowledged
PF-2	Sandwich attack	🟡	**MEDIUM**	Acknowledged
SLT-1	Incorrect Design	🟡	**MEDIUM**	Acknowledged
SLT-2	Missing key functions	🟡	**MEDIUM**	Resolved
GLOBAL-2	Incorrect Design	🟡	**MEDIUM**	Acknowledged
OB-6	Blacklisted Token	🟢	**LOW**	Acknowledged
SVAU-8	CEI Pattern	🟢	**LOW**	Resolved
BU-1	Phantom addresses	🟢	**LOW**	Acknowledged
BS-1	Incorrect Validation	🟢	**LOW**	Resolved
SXLPM-2	Bypass check	🟢	**LOW**	Acknowledged
SLT-3	Incorrect Design	🟢	**LOW**	Resolved
GLOBAL-3	Incorrect Design	🟢	**LOW**	Acknowledged
GLOBAL-4	Incorrect Validation	🟢	**LOW**	Resolved
GLOBAL-5	Griefing	🟢	**LOW**	Acknowledged
GLOBAL-6	Incorrect Design	🟢	**LOW**	Acknowledged
GLOBAL-7	Incorrect Design	🟢	**LOW**	Acknowledged
SXLPM-3	Informational	🔵	**INF**	Resolved
SXLPM-4	TYPO	🔵	**INF**	Resolved
GLOBAL-8	Architectural Error	🔵	**INF**	Partially Resolved

OB-1 | Orders can be executed with stale prices

Severity	Category	Status
● HIGH	Price Manipulation	Partially Resolved

Description of the issue

Currently when creating and executing orders from the order book, it is up to the caller to specify which Pyth price feed gets updated in order to update the prices of the assets the user is interacting with.

As seen in the following Pyth implementation, at address: `0xe9d69CdD6Fe41e7B621B4A688C5D1a68cB5c8ADc` in arbiscan:

```
function updatePriceFeeds(
        bytes[] calldata updateData
    ) public payable override{}
```

Once you call `updatePriceFeeds` you have to pass an array of bytes. If this array is empty, the function will simply execute without failure and not ask for any fee as `requiredFee` will be 0:

```
uint requiredFee = getTotalFee(totalNumUpdates);
if (msg.value < requiredFee) revert PythErrors.InsufficientFee();
```

Therefore, if you call any of the affected functions: `executeSwapOrder`, `executeIncreaseOrder`, and `executeDecreaseOrder` with an empty array of updateData, no prices will be updated, using stale prices from Pyth.

This can be done to profit from using stale prices in swaps for example. Users can exploit the fact that Pyth is not updated very frequently in Arbitrum to access stale prices by providing an empty array and profiting from it in order executions. This issue is also found in other contracts like Xlp Manager, SXlp Manager, Reward Router, and Reward Router V2.

Recommendation

Consider creating a mapping of price feeds and map the corresponding index/collateral token to a price feed to ensure that the correct price feed gets updated.

Resolution

The recommended fix as not been implemented. Though, as in a different issue the isLeverageEnabled variable has been shifted to false, this should not be an issue for users right now. The protocol can still act maliciously or gets their keys licked and send an empty array to not update the price. Though the MAX_AGE also as been shortened, which makes this scenario much harder to happen.

OB-2 | Incorrect requirement to send execution fee when creating an order

Severity	Category	Status
🔴 HIGH	Incorrect Operators	Resolved

Description of the issue

In the order book, you can create both increase and decrease orders using the `createDecreaseOrder` and `createIncreaseOrder` functions.

Both functions require the user to send an execution fee to reward the keepers for executing the transaction.
There is a mistake when assigning the comparison operators in the require statement:

```
require(msg.value > minExecutionFee,
"OrderBook: insufficient execution fee");
```

As seen in the code, it requires that the amount of ether sent is greater than the minimum execution fee, while it should be equal to or greater.
This will end up costing users in the long run since they will always have to pay more than the minimum requirement.

POC:

```
  it.only("Incorrect requirement to send execution fee when creating an
order", async () => {

    await positionManager.setOrderKeeper(user1.address, true)

    await positionManager.connect(user1).increasePosition([dai.address],
btc.address, expandDecimals(50000, 18), 0, toUsd(100000), false,
toNormalizedPrice(50000))

    await router.connect(user1).approvePlugin(orderBook.address)

    let executionFee = orderBook.minExecutionFee();

    await orderBook.connect(user1).createDecreaseOrder(
      btc.address, // indexToken
      toUsd(10000), // sizeDelta
      dai.address, // collateralToken
      toUsd(5000), // collateralDelta
      false, // isLong
      toUsd(0), // triggerPrice
      true, // triggerAboveThreshold
      {value: executionFee}
    );

    let orderIndex = (await orderBook.decreaseOrdersIndex(user1.address)) -
1

    expect(await
positionManager.connect(user1).executeDecreaseOrder(user1.address,
orderIndex, user1.address)).to.be.revertedWith("OrderBook: insufficient
execution fee");
  })
```

Recommendation

Add an equal sign to the require:

```
require(msg.value >= minExecutionFee,
"OrderBook: insufficient execution fee");
```

Resolution

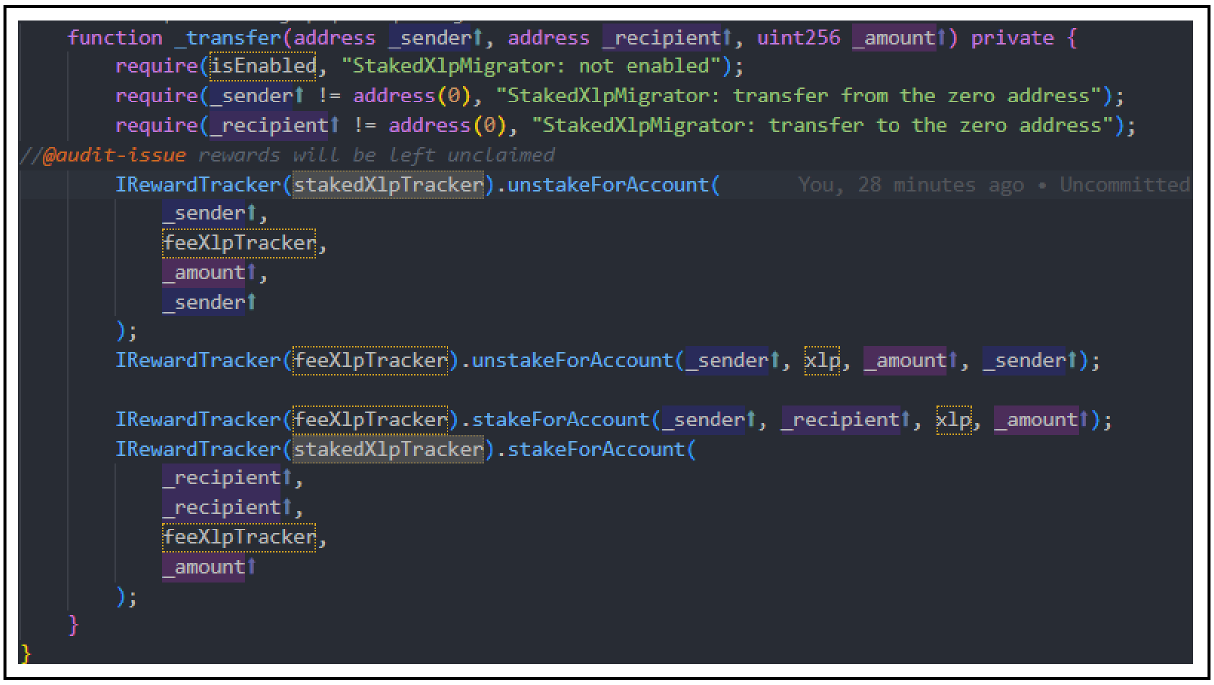
Fixed

SXLPMG-1 | XLP is not compounded when migrating

Severity	Category	Status
● HIGH	Uncompounded Rewards	Resolved

Description of the issue

When calling the `transfer` function from the XLP migrator contract, XLP is unstaked from account1 and staked for account2. The issue is that the rewards are left unclaimed/uncompounded before unstaking.



```
    function _transfer(address _sender, address _recipient, uint256 _amount) private {
        require(isEnabled, "StakedXlpMigrator: not enabled");
        require(_sender != address(0), "StakedXlpMigrator: transfer from the zero address");
        require(_recipient != address(0), "StakedXlpMigrator: transfer to the zero address");
//@audit-issue rewards will be left unclaimed
        IRewardTracker(stakedXlpTracker).unstakeForAccount(        You, 28 minutes ago • Uncommitted
            _sender,
            feeXlpTracker,
            _amount,
            _sender
        );
        IRewardTracker(feeXlpTracker).unstakeForAccount(_sender, xlp, _amount, _sender);

        IRewardTracker(feeXlpTracker).stakeForAccount(_sender, _recipient, xlp, _amount);
        IRewardTracker(stakedXlpTracker).stakeForAccount(
            _recipient,
            _recipient,
            feeXlpTracker,
            _amount
        );
    }
}
```

Recommendation

Call the `_compound()` function from the reward router v2 contract before unstaking.

```
function _compound(address _account) private {
        _compoundGmx(_account);
        _compoundGlp(_account);
    }
```

Resolution

Fixed

SPR-1 | Synthetic Does Not Allow Swapping

Severity	Category	Status
● HIGH	Incorrect Design	Resolved

Description of the issue

Users are allowed to provide a path length of either 1 or 2. If the path length is 2, the function `_swap` will be called instead of swapping token1 for token2. The transaction will revert since Lexer doesn't allow for swapping in synthetic mode. In the event that the user ends up calling the `_swap` function, their transaction will revert and the order will need to be canceled this will end up costing users additional gas.

Recommendation

Do not allow users to provide a path with a length of 2.

Resolution

Fixed, swapping has been removed completely.

SPR-2 | Orders executed from the Routers forget to update the prices

Severity	Category	Status
● HIGH	Stale prices	Partially Resolved

Description of the issue

Orders directly executed from the SPositionRouter and the PositionRouter are not updating the price of the assets that are going to be traded. When calling `executeIncreasePosition` and ` executeDecreasePosition` the call to update the prices `_updatePrices(_priceUpdateData);` is missing.

Therefore, if an order is executed directly from the routers, it will either fail to be executed due to too stale prices or be executed with a wrong and un-updated price.

Recommendation

Do call ` updatePrices(_priceUpdateData); ` in every function that fetches the price of an asset in the routers directly.

Resolution

Partially Fixed. The recommendation has not been followed. Instead, the execute functions do check that the caller is ` address(this)`, to know if the execution was a batch order execution initiated by Lexer. If not, the price is updated and checks that the price was indeed updated on that block. There might be some edge cases when the array has incorrect data, and the price might be slightly off by that block. Overall, the fix looks good, having in mind a possible edge case.

VAU-1 | Swaps performed directly from the Vault will use stale prices and can be DOS'ed

Severity	Category	Status
● HIGH	Stale Prices	Acknowledged

Description of the issue

Generally, the Vault is called from the order book contract although the Vault is also structured so it can be called directly.

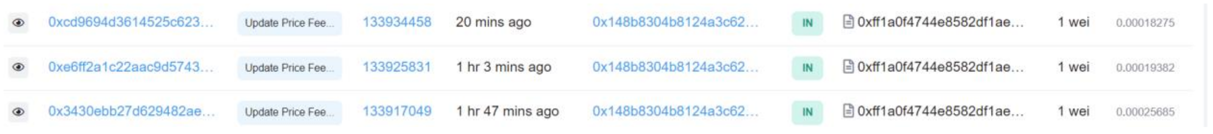
When interacting with the vault directly, as Lexer did shift from using Chainlink oracles to Pyth, the price will not be updated.
Users can perform swaps with stale prices to their favor to drain the vault.

In addition, the swap functionality can also be DOS for swaps where the price is older than the specified in `getPriceNoOlderThan()`:

```
pyth.getPriceNoOlderThan(priceID, maxPriceAge);
```

This function returns the oldest price that is stored and it must not be older than the age specified.

As seen in the following Pyth proxie, which can be found at address: `0xff1a0f4744e8582DF1aE09D5611b887B6a12925C` in arbiscan:



👁	0xcd9694d3614525c623...	Update Price Fee...	133934458	20 mins ago	0x148b8304b8124a3c62...	IN	0xff1a0f4744e8582df1ae...	1 wei	0.00018275
👁	0xe6ff2a1c22aac9d5743...	Update Price Fee...	133925831	1 hr 3 mins ago	0x148b8304b8124a3c62...	IN	0xff1a0f4744e8582df1ae...	1 wei	0.00019382
👁	0x3430ebb27d629482ae...	Update Price Fee...	133917049	1 hr 47 mins ago	0x148b8304b8124a3c62...	IN	0xff1a0f4744e8582df1ae...	1 wei	0.00025685

prices can be more than one hour without being updated. Most likely if this happens, the maximum age from the function `getPriceNoOlderThan` will be smaller than 1 hour and the transaction will revert because the call to `getPriceNoOlderThan` reverts with a StalePriceError if the on-chain price is from more than age seconds in the past.

Recommendation

Do call `updatePrices()` in every function that fetches the price from the pyth oracle in the vault.

Resolution

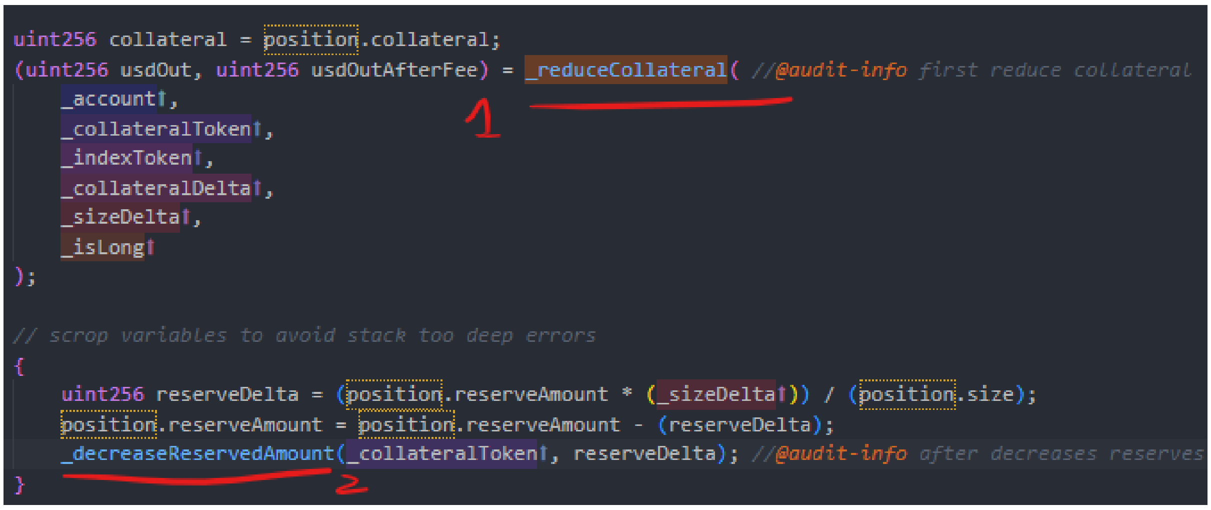
The maxAge will be narrowed to values approximately between the 5 and 8 seconds. We consider the issue as acknowledged as even though it might help, the price might still be a few seconds stale.

SVAU-1 | Incorrect order of function calls might DOS liquidations

Severity	Category	Status
● HIGH	DOS	Resolved

Description of the issue

Currently in SVault, when decreasing a position by calling `_decreasePosition()`, the `_reduceCollateral()` function is called first and then it is followed by the `_decreaseReservedAmount()` call.



```
uint256 collateral = position.collateral;
(uint256 usdOut, uint256 usdOutAfterFee) = _reduceCollateral( //@audit-info first reduce collateral
    _account,
    _collateralToken,
    _indexToken,
    _collateralDelta,
    _sizeDelta,
    _isLong
);

// scrop variables to avoid stack too deep errors
{
    uint256 reserveDelta = (position.reserveAmount * (_sizeDelta)) / (position.size);
    position.reserveAmount = position.reserveAmount - (reserveDelta);
    _decreaseReservedAmount(_collateralToken, reserveDelta); //@audit-info after decreases reserves
}
```

Both of the functions share a state which is the mapping `reservedAmounts[_token]`
.
The function named reduce collateral makes a check validating that the amount reserved of the collateral token is less or equal to the amount of collateral in the pool.

```
function _decreasePoolAmount(address _token, uint256 _amount) private {
        _validate(poolAmounts[_token] >= _amount, 59);
        poolAmounts[_token] = poolAmounts[_token] - _amount;
        _validate(reservedAmounts[_token] <= poolAmounts[_token], 50);

        // console.log(poolAmounts[_token]);
        emit DecreasePoolAmount(_token, _amount);
    }
```

On the other hand, the _decreaseReservedAmount function does reduce the amount of reserved collateral.

```
function _decreaseReservedAmount(address _token, uint256 _amount)
private {
       _validate(reservedAmounts[_token] >= _amount, 57);
       reservedAmounts[_token] = reservedAmounts[_token] - (_amount);
       emit DecreaseReservedAmount(_token, _amount);
   }
```

Therefore, as a simple explanation, the collateral from the pool is subtracted before the amount of collateral in the reserve. This itself is already wrong on its own, but the severity increases in regards to the previously mentioned check:

```
_validate(reservedAmounts[_token] <= poolAmounts[_token], 50);
```

which is much more prone to fail due to decreasing the pool amounts before the reserves. This will halt any functionality that calls the `_decreasePosition()` function, such as liquidations.

Recommendation

Do shift the order in which those functions are called. First call `_decreaseReservedAmount()` and then `_reduceCollateral()`

Resolution

Fixed

SVAU-2 | If isManagerMode is disabled in vault, XLP won’t be burned, and several states won’t be updated.

Severity	Category	Status
🔴 HIGH	Loss of funds	Partially Resolved

Description of the issue

In functions like `buyUsdg()` and `sellUsdg()` from the vault, there is a check that allows users to call the vault directly if isManagerMode is de-activated.

```
function _validateManager() private view {
        if (inManagerMode) {
            _validate(isManager[msg.sender], 54);
        }
    }
```

If this happens after having isManagerMode turned on, users that added liquidity and bought usdg from the SXlpManager, will not get their Xlp burned, and several states will be un-updated. Allowing the user, to withdraw again through the SXlpManager because his Xlp is not burned. Furthermore, buying usdg from the vault directly, does not have a slippage control, in comparison with calling it from the manager, which it does have it.

Recommendation

Do not de-activate `isManagerMode` so this scenario is not given.

Resolution

There is no change on the smart contract level, but ` isManagerMode` will not be disabled.

GLOBAL-1 | Missing refund of the execution fee to the user

Severity	Category	Status
● HIGH	Loss of funds	Acknowledged

Description of the issue

Lexer requires users to send an execution fee in ether when creating any type of order that has to be relayed by keepers. This happens in several contracts from the codebase, therefore, it is best to include this issue as a global risk. Whatever execution fee the user sends will never be reimbursed, potentially resulting in significant losses for users and protocols integrating with Lexer.

Recommendation

Consider adopting the GMX V2 model of calculating the execution cost + callback gas limits using `gasLeft()` and accounting for the 63/64 (EIP-150) rule in external calls.

Resolution

Acknowledged

MH-1 | Inefficient architecture

Severity	Category	Status
🔴 HIGH	Incorrect Design	Resolved

Description of the issue

The Migration handler is a contract that allows you to redeem and sell your usdg and swap them for another token. This contract overall is quite poorly written and has several mistakes:

- Hardcoded swap paths which may not be the most liquid or optimal ones in the future.
- Deadline as block.timestamp which allows miners to hold the transaction.
- Usage of approve instead of forceApprove which may revert on non-zero allowances
- Messy architecture with a lot of repeated and unnecessary nsloc

Recommendation

Most of the contract logic would be able to be replaced by using an aggregator like 1inch. Redeem your usdg and directly make an external call to 1inch following their documentation closely. Understand that 1inch uses several off-chain calculations, therefore, extra infrastructure should be built when interacting with it.

Resolution

Fixed

VAU-2 | Stale cumulativeFundingRate is being used when fetching validateLiquidation externally which allows unhealthy positions to appear healthy

Severity	Category	Status
● HIGH	Late Liquidations	Resolved

Description of the issue

Any system relying on the external `validateLiquidation()` function to determine if their positions or users' positions are liquidatable can receive an incorrect response. This is because `validateLiquidation` does not update the CumulativeFundingRate.

If a significant amount of time has passed since the last call of `updateCumulativeFundingRate`, `validateLiquidation` can return a false state. As demonstrated in the following PoC, the user's position may appear healthy when calling `validateLiquidation`, but when `updateCumulativeFundingRate` is called, the position becomes liquidatable. This can be highly misleading for users or developers building on top of Lexer who rely on the public view function to check the state of users' positions.

POC:

```
it.only("validateLiquidation can return incorrect data", async () => {
    await shortsTracker.setIsGlobalShortDataReady(true)
    await glpManager.setShortsTrackerAveragePriceWeight(10000)
    expect(await
glpManager.shortsTrackerAveragePriceWeight()).to.be.equal(10000)

    await positionManager.connect(user1).increasePosition([dai.address],
btc.address, expandDecimals(50000, 18), 0, toUsd(100000), false,
toNormalizedPrice(60000))

    await btcPriceFeed.setLatestAnswer(toChainlinkPrice(48000))
    await positionManager.connect(user0).increasePosition([dai.address],
btc.address, expandDecimals(10000, 18), 0, toUsd(100000), false,
toNormalizedPrice(48000))
```

```
    await increaseTime(provider, 86400 * 365)
    await btcPriceFeed.setLatestAnswer(toChainlinkPrice(49000))

    let [liquidationStateBefore, marginFeeBefore] = await
vault.validateLiquidation(user0.address, dai.address, btc.address, false,
false)
    expect(liquidationStateBefore).to.eq(BigNumber.from("0"));


expect(marginFeeBefore).to.eq(BigNumber.from("100000000000000000000000000000
0000"));

    await vault.updateCumulativeFundingRate(dai.address, btc.address)

    let [liquidationStateAfter, marginFeeAfter] = await
vault.validateLiquidation(user0.address, dai.address, btc.address, false,
false)

expect(marginFeeAfter).to.eq(BigNumber.from("7816666666666666666666666666666
667"));
    expect(liquidationStateAfter, "liquidation state").to.be.eq(1)


    const [liquidationState, marginFee] = await
getLiquidationState(user0.address, dai.address, btc.address, false, false)
    expect(liquidationState, "liquidation state").to.be.eq(1)

    await positionManager.setLiquidator(user1.address, true)
    await positionManager.connect(user1).liquidatePosition(user0.address,
dai.address, btc.address, false, user1.address)
 })
```

Recommendation

Make sure to document this behavior.

Resolution

Fixed by adding `updateCumulativeFundingRate(_indexToken);` to update the cumulative funding rate

OB-3 | Pyth fee is currently paid by the keeper's

Severity	Category	Status
● MEDIUM	Loss of funds	Partially Resolved

Description of the issue

In the orderbook, when orders are being executed, the caller has to send a small amount of ether that behaves as part of the fee that is needed to execute an update pyth's price feed. Currently, the fee to update pyth's price feed is not accounted for inside the minimum execution fee that the user has to send when creating an order, therefore the keepers could end up running a deficit.

Recommendation

Make sure to include in the calculation of the execution fee the extra amount that will cost to update the pyth price feed.

Resolution

It is not enforced on the smart contract level but it will be enforced by internal calculations from the team.

OB-4 | Keeper griefing when executing orders with WETH as collateral

Severity	Category	Status
● MEDIUM	Griefing	Resolved

Description of the issue

In the order book, when orders are being executed with WETH as the collateral token, a callback is made to the user's address that unwraps the WETH and sends ETH to the user.

```
_transferOutETH(_amountOut, payable(order.account));
```

The user can be a smart contract that burns/wastes most of the gas in order to make the overall gas consumption more than what the keeper is anticipating. This could end up causing the keeper to run a deficit over and over again, draining the keeper's reserves.

Recommendation

Consider forwarding a specific amount of gas with the external call so that it is not possible to spend more than the said amount. Also, consider using an assembly low-level call to avoid copying the returned data from the callback contract to memory.

```
assembly { success := call(gasLimit, receiver, amount, 0, 0, 0, 0) }
```

Resolution

Fixed

OB-5 | Missing validation when decreasing orders could cause losses in the pool

Severity	Category	Status
● MEDIUM	Griefing	Acknowledged

Description of the issue

There is nothing stopping a malicious actor from decreasing their position by 99.9% and leaving a small amount of wei left over. The gas price for liquidating the position can be larger than the position itself, which works against the liquidators and could cause significant losses to the pool since the liquidationFeeUsd is taken from the pool in the event that the position isn't large enough to cover it. This issue can also be found in the position router and its synthetic variation.

Recommendation

Consider closing a position if a user reduces their position to the point where they will not be able to cover the liquidationFeeUsd.

Resolution

Acknowledged

PR-1 | Execution cost + Callback gas limit should never surpass the minExecutionFee reimbursed to the keepers.

Severity	Category	Status
● MEDIUM	Incorrect Validation	Partially Resolved

Description of the issue

When executing orders it is never enforced that the execution cost + the callback gas limit for every contract is less or equal to the execution fee sent by the user, which would make losses for keepers and allow griefing by passing whitelisted contracts with high custom gas limits.

Recommendation

Do make internal calculations and add a check and reinforcements so that the cost of executing the functions `executeIncreasePosition()` and `executeDecreasePosition()` and the gas cost spent on the callback from both functions is never more than the minExecutionFee. By calculating beforehand we also mean to not add very expensive callback gas limits to any of the customCallbackGasLimits targets.

```
uint256 _customCallbackGasLimit =
customCallbackGasLimits[_callbackTarget];
```

This is because anyone could basically grief the keeper for free by passing an address that is included in the customCallbackGasLimits mapping and exceed the execution fee.

Resolution

Not fixed at the smart contract level but it will be enforced by internal calculations from the team.

PR-2 | Keepers can steal the execution fee from a user by canceling a user’s position

Severity	Category	Status
● MEDIUM	Malicious Keeper	Acknowledged

Description of the issue

Currently, the only users who can cancel positions are the same users who created those positions or a keeper. When canceling a position, the caller can specify the receiver of the previously sent execution fee when the position was created.

If any keeper is compromised or misbehaves, orders can be canceled for profit for the keeper and loss for the user.

Recommendation

Make sure to document this behavior, so users are aware of such risks.

Resolution

Acknowledged

PR-3 | Users or protocols interacting with the router can be DOS'ed by keepers

Severity	Category	Status
🟡 MEDIUM	Malicious Keeper	Acknowledged

Description of the issue

Users or protocols integrating with the position router contract can specify a callback contract when creating a position. This callback is then triggered when the order is relayed or canceled by the keepers.

A malicious/compromised keeper can make those callbacks fail due to a lack of gas.

In the functions where the callback is triggered, this one is wrapped inside a try-catch making an external call to the contract the user who created the position specified at the beginning:

```
bool success;
       try
IPositionRouterCallbackReceiver(_callbackTarget).gmxPositionCallback{
gas: _gasLimit }(_key, _wasExecuted, _isIncrease) {
           success = true;
       } catch {}
```

Malicious keepers can make the callback fail by calculating the exact amount of gas it would take for the external call to run out of gas.

This is possible thanks to the EIP-150 and the 63/64 rule where 63/64 of the remaining gas is forwarded with the external call, and if there is a revert on execution there, you still have 1/64 gas to finish the execution of the current function.

As the callback is inside a try-catch that does not revert when the external call runs out of gas, the keeper can calculate the exact gas they would need to forward at the beginning of the transaction (accounting for the gas spent in the callback contract too) so that the external call runs out of gas and the callback can never be performed.

The attack would look something like this:

Malicious keeper calculates:
- The gas costs of the execution of the whole function (without the external call): 1M gas

- The gas cost of the callback contract: 200k gas

- As there is already specified a _gasLimit forwarded, which refers to the callback gas limit, it should be bigger than the gas spent of the callback contract: 300k gas

Attack:
- Keeper calls the function to execute the position forwarding 1.1M gas
- External call to the callback contract will forward around 120k gas because there is still events to be emitted and the execution is not yet finished.

- The external call will fail due to lack of gas (120k gas forward vs 200k gas needed)

- The transaction does not revert due to the catch statement not reverting, and the execution will finish with the gas left.

Recommendation

Document this behavior so users are aware of such risks.

Resolution

Acknowledged

SPR-3 | Gas bomb attack on external call

Severity	Category	Status
🟡 MEDIUM	Gas bomb	Resolved

Description of the issue

The function `_transferOutETHWithGasLimitFallbackToWeth` makes a keeper vulnerable to a payload attack where a malicious user creates a large gas bomb by loading a ton of return data into memory. Due to how Solidity handles return data, this malicious payload will be loaded into memory, which will end up costing the keeper additional gas that wasn't accounted for in the executionFee. This can lead to the keeper running a deficit.

Recommendation

Use a low-level call to avoid loading the return data into memory.

```
assembly { success := call(gasLimit, receiver, amount, 0, 0, 0, 0) }
```

Resolution

Fixed

SPR-4 | Users will not be able to cancel orders if isLeverageEnabled is false

Severity	Category	Status
🟡 MEDIUM	DOS	Partially Resolved

Description of the issue

In the event that `isLeverageEnabled` is set to false in the contract `positionRouter`, users will no longer be able to call the functions `cancelIncreasePosition` and `cancelDecreasePosition` due to a check in `_validateExecutionOrCancellation`.

```
 (!isLeverageEnabled && !isKeeperCall) { revert("403"); }
```

This will make it impossible for a user to cancel their order without having a keeper do it for them which could end with either the keeper or the user losing out on the executionFee the user paid upon creation.

POC:

```
it.only("POC: Users will not be able to cancel orders if isLeverageEnabled
is false ", async () => {
    await dai.mint(user1.address, expandDecimals(600000, 18))

    await vault.connect(user1).addRouter(user1.address)
    await dai.connect(user1).transfer(vault.address, expandDecimals(60000,
18))
    await vault.buyUSDG(dai.address, user1.address)

    await timelock.setContractHandler(positionRouter.address, true)

    await dai.mint(user1.address, expandDecimals(60000, 18))

    await dai.connect(user1).approve(router.address, expandDecimals(60000,
18))

    let params = [
      [dai.address], // _path
      bnb.address, // _indexToken
      expandDecimals(6000, 18), // _amountIn
      expandDecimals(1, 18), // _minOut
      expandDecimals(7000000000000000, 18), // _sizeDelta
      false, // _isLong
      toUsd(300), // _acceptablePrice
    ]

    const referralCode =
"0x0000000000000000000000000000000000000000000000000000000000000123"

    await router.addPlugin(positionRouter.address)
    await router.connect(user1).approvePlugin(positionRouter.address)

    await positionRouter.setPositionKeeper(positionKeeper.address, true)

    await dai.mint(user1.address, expandDecimals(600, 18))

    await
positionRouter.connect(user1).createIncreasePosition(...params.concat([4000,
referralCode, AddressZero]), { value: 4000 })

    let key = await positionRouter.getRequestKey(user1.address, 1)

    await positionRouter.setAdmin(user0.address)

    await positionRouter.connect(user0).setIsLeverageEnabled(false)

    expect(await positionRouter.connect(user1).cancelIncreasePosition(key,
user1.address)).to.be.revertedWith("403");
  })
```

Recommendation

Make sure to allow users to cancel orders.

Resolution

Not fixed at the smart contract level but they will strictly make sure the `isLeverageEnabled` set to true.

SPR-5 | Users are unable to close orders when maxProfit is hit

Severity	Category	Status
● MEDIUM	DOS	Acknowledged

Description of the issue

In the event that a user earns profits equivalent to the maxProfit they will no longer be able to close their position by themselves due to the check placed in the function `validateliquidation`

```
If(hasProfit && delta >= _vault.tokenToUsdMin(_collateralToken,
position.reserveAmount)) {
            if (_raise) { revert("Vault: maxProfit hit"); }
            return (3, marginFees);
        }
```

The user will have to wait for a keeper to execute a liquidation order before being able to retrieve the profits that they have earned. This can result in the user not getting the outcome they wanted.

Recommendation

Make sure to document this behavior so users are aware of this behavior.

Resolution

Acknowledged. The functionality will be documented to users, so they are aware.

SPR-6 | Users are allowed to create increase orders with non whitelisted tokens

Severity	Category	Status
● MEDIUM	Incorrect Validation	Resolved

Description of the issue

Missing token validation allows users to create and increase orders with non-whitelisted tokens. However, these orders can never be executed due to the 'validatetoken' check in the vault/SVault contracts. As a result, this will end up costing users extra gas since they will have to cancel their order after creating the order.

Recommendation

Check if the user is using whitelisted tokens when creating the increase order.

Resolution

Fixed

SPR-7 | Dust positions can be difficult to liquidate

Severity	Category	Status
● MEDIUM	Griefing	Acknowledged

Description of the issue

A malicious user can create orders only worth a few wei if they interact with the protocol through the positionRouter since its missing the check that is placed in orderBook.createIncreaseOrder:

```
require(
  _purchaseTokenAmountUsd >= minPurchaseTokenAmountUsd,
  "OrderBook: insufficient collateral"
      );
```

This would make it act against the liquidators, as the gas price of liquidating the position can be larger than the position itself. Moreover, if the position is not substantial enough to cover it, the liquidationFeeUsd could result in significant losses for the pool as it is deducted directly from it.

Recommendation

Consider implementing this ´minPurchaseTokenAmountUsd´ check in the position router.

Resolution

Acknowledged

SVAU-3 | MAX_FUNDING_RATE_FACTOR is never enforced in SVault

Severity	Category	Status
● MEDIUM	Broken Invariant	Resolved

Description of the issue

Lexer has several invariants and values that should never be surpassed. One of them is MAX_FUNDING_RATE_FACTOR which is set to 2%. Said variable is never checked against, allowing higher and unhealthy values to be reached. Having a too high funding rate factor can definitely affect the long-term healthiness of the protocol.

Recommendation

Do enforce that MAX_FUNDING_RATE_FACTOR can never be surpassed.

Resolution

Fixed

SVAU-4 | Accounting can be disrupted

Severity	Category	Status
● MEDIUM	Accounting manipulation	Resolved

Description of the issue

A malicious user can disrupt the accounting by being the very first user to interact with the system and call the vault contract directly to increase an order instead of interacting via the router. This can be done because `isLeverageEnabled` is initially set to true and only changed to false after the first user interacts with the position router. The user could create a short that would not call the function 'updateGlobalShortData' and could bypass 'maxGlobalSize'.

Recommendation

Monitor closely when deploying the vault contract.

Resolution

Fixed

SVAU-5 | Gasprice Is Not Validated

Severity	Category	Status
● MEDIUM	Incorrect Validation	Resolved

Description of the issue

The validation set in place originally by GMX was done to make sure that minExecutionFee is enough to cover the gas cost of the execution. Currently, it has been removed. This can lead to the keeper having to pay more than the executionFee that was given if the gas price rises.

Recommendation

Reimplement the `_validateGasPrice`

Resolution

Fixed

SVAU-6 | Funding fees can be gamed

Severity	Category	Status
🟡 MEDIUM	Incorrect Design	Acknowledged

Description of the issue

Funding fees are kept track of in intervals. This can be gamed by a user waiting until the very end of an interval and decreasing their position, or a user buying and selling within the same funding interval, which will result in no funding fees being paid.

Recommendation

Monitor closely and adjust the funding intervals accordingly.

Resolution

Acknowledged

SVAU-7 | setTokenConfig Doesn't Check If a Token Is Used

Severity	Category	Status
● MEDIUM	Incorrect Validation	Acknowledged

Description of the issue

There is no check in place to ensure that a token is not in use before calling the function 'setTokenConfig', which can override the values currently set. This could come as an unexpected surprise for users who can be affected by changes like 'maxProfitBasisPoints', which will cap the user's profits at a lesser profit margin.

Recommendation

Consider checking if a token is in use before allowing setTokenConfig to be called.

Resolution

Acknowledged

SXLPM-1 | Missing max cap on WhitelistedTokens

Severity	Category	Status
● MEDIUM	Incorrect Validation	Acknowledged

Description of the issue

There is no cap on the amount of whitelisted tokens that can be added via the `setTokenConfig` function. If too many tokens are added, it could lead to a DOS of the `getAum` function since `getAum` loops over all whitelisted tokens in the `allWhitelistedTokens` array.

```
function getAum(bool maximise) public view returns (uint256) {
   uint256 length = vault.allWhitelistedTokensLength();
   uint256 aum = aumAddition;
   uint256 userProfits = 0;
   ISVault _vault = vault;

for (uint256 i = 0; i < length; i++) {
```

Recommendation

Consider adding a cap to how many whitelisted tokens should be allowed

Resolution

Acknowledged

VPF-1 | Price Manipulation possible if isSecondaryPriceEnabled set to false

Severity	Category	Status
● MEDIUM	Price Manipulation	Partially Resolved

Description of the issue

In the event that `isSecondaryPriceEnabled` is set to false, the protocol uses the `getAmmPrice` function to retrieve the price based on a PancakePair's reserves, which can be highly susceptible to manipulation.

```
function getPriceV1(address _token, bool _maximise, bool
_includeAmmPrice) public view returns (uint256) {
uint256 price = getPrimaryPrice(_token, _maximise);
if (_includeAmmPrice && isAmmEnabled) {
  uint256 ammPrice = getAmmPrice(_token);
  if (ammPrice > 0) {
  if (_maximise && ammPrice > price) {
     price = ammPrice;
     }
  if (!_maximise && ammPrice < price) {
     price = ammPrice;
     }
  }
}
if (isSecondaryPriceEnabled) {
  price = getSecondaryPrice(_token, price, _maximise);
}
```

Additionally, when isSecondaryPriceEnabled is set to false the functions ´getPrimaryPrice´ and ´getAmmPrice´ will be called everytime a user increases/decreases their position which will spend unnecessary gas because it will be overwritten by ´getSecondaryPrice´

Recommendation

Great care should be taken when considering to set isSecondaryPriceEnabled to false.

Resolution

Partially Resolved. It is not enforced on the smart contract level but it will be enforced by the team by always setting `isSecondaryPriceEnabled` to true.

PF-1 | PriceFeed uses the deprecated chainlink interface

Severity	Category	Status
● MEDIUM	Deprecated function	Acknowledged

Description of the issue

According to Chainlink’s documentation (API Reference), the latestAnswer function is deprecated. This function does not throw an error if no answer has been reached, but instead returns 0, possibly causing an incorrect price to be fed to the different price feeds or even a Denial of Service by a division by zero.

```
function latestAnswer() public override view returns (int256) {
        return answer;
  }
```

Recommendation

Either don’t use PriceFeed.sol because you are integrating with pyth already, or shift the interface to the new one using latestRoundData() instead.

Resolution

Acknowledged

PF-2 | PriceFeed is sandwichable at price updates

Severity	Category	Status
● MEDIUM	Sandwich attack	Acknowledged

Description of the issue

The way PriceFeed.sol has been designed allows to perform sandwich attacks on price updates. When they set the price in setLatestAnswer, you can sandwich the previous price with any funciton calling getRoundData for profit.

```
function setLatestAnswer(int256 _answer) public {
       require(isAdmin[msg.sender], "PriceFeed: forbidden");
       roundId = roundId + 1;
       answer = _answer;
       answers[roundId] = _answer;
   }

   // returns roundId, answer, startedAt, updatedAt, answeredInRound
   function getRoundData(uint80 _roundId) public override view
       returns (uint80, int256, uint256, uint256, uint80)
   {
       return (_roundId, answers[_roundId], 0, 0, 0);
   }
```

Recommendation

If PriceFeed.sol is not going to be used, it should be deleted.

Resolution

Acknowledged

SLT-1 | Approvals on non-zero allowance will revert

Severity	Category	Status
● MEDIUM	Incorrect Design	Acknowledged

Description of the issue

In the timelock contract, there are functions to signal approvals and to execute them. When executing an approval, the approve function is used without accounting for previous allowances. Approvals to addresses to which allowances are more than 0, will fail.

```
function approve(address _token, address _spender, uint256 _amount)
external onlyAdmin {
       bytes32 action = keccak256(abi.encodePacked("approve", _token,
       _spender, _amount));
       _validateAction(action);
       _clearAction(action);
       IERC20(_token).approve(_spender, _amount);

    }
```

Recommendation

Migrate to use the new forceApprove function from OZ.

Resolution

Acknowledged

SLT-2 | Missing key functions in timelock.

Severity	Category	Status
● MEDIUM	Missing key functions	Resolved

Description of the issue

The vault and svault contracts do have a list of functions that will be managed though the timelocks (SLexTimelocks and Timelock). These functions stand out by having a _onlyGov() modififer:

```
    function setInManagerMode(bool _inManagerMode) external override {
        _onlyGov();
        inManagerMode = _inManagerMode;
    }
```

Therefore, the functions carrying this modifier should be added to the timelock so governance can process and execute updates.

Currently, the following functions are missing in the timelock, not allowing them to be updated through governance.

```
setErrorController(), setInManagerMode(), clearTokenConfig(),
upgradeVault(),
```

Recommendation

Add those functions to the timelocks. If signaling functions are also necessary for some of them, do also include them accordingly.

Resolution

Fixed

GLOBAL-2 | Stock Splits are unaccounted for

Severity	Category	Status
● MEDIUM	Incorrect Design	Acknowledged

Description of the issue

Lexer aims to enable synthetic assets, which can include equities. However, it's important to note that supporting equities comes with potential risks. One such risk is presented by forward stock splits, where a stock's price is halved, and the number of shares a user owns is doubled. Similar scenarios can also arise from reverse stock splits, mergers, and acquisitions.

Recommendation

If equities are used, monitor for such behavior.

Resolution

Acknowledged

OB-6 | Blacklisted tokens can halt liquidations.

Severity	Category	Status
● LOW	Blacklisted Token	Acknowledged

Description of the issue

A user can create a position in the Lexer protocol and if that same user gets blacklisted from the backing token (USDC) after creating the position, the position will not be able to be liquidated until the user's position incurs a 100% loss. This is because any transfer of the backing token (USDC) would fail during the transfer process.

Recommendation

While in this case a blacklisted account is not a critical harm for the protocol, the reasoning why this is an issue on the first place is that Lexer is using a push pattern instead than a pull pattern and it should be (PULL over PUSH).
Be aware that some tokens have the ability to blacklist users from interacting with them and if willing to, refactor the logic so tokens can be claimed instead of directly sent when liquidating.

Resolution

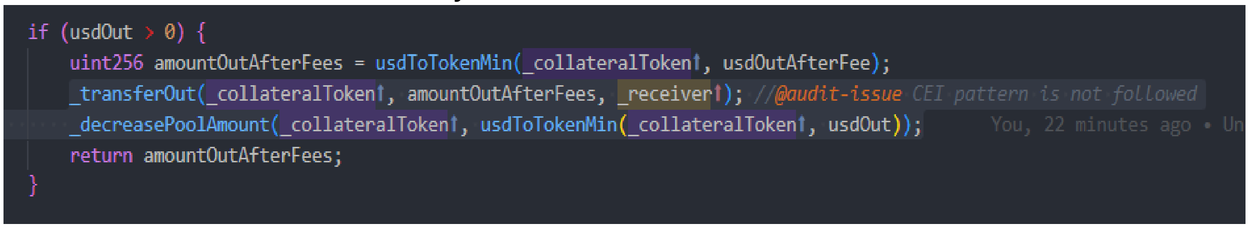
Acknowledged

SVAU-8 | Checks-effects-interactions pattern is not being adhered to

Severity	Category	Status
● LOW	CEI Pattern	Resolved

Description of the issue

Within the `_decreasePosition()` function, the transfers are executed before the state variable changes happen. First, the collateral token is transferred out from the vault and then the pool amount is decreased. This violates the CEI pattern and makes the contract vulnerable to reentrancy attacks.



```
if (usdOut > 0) {
    uint256 amountOutAfterFees = usdToTokenMin(_collateralToken, usdOutAfterFee);
    _transferOut(_collateralToken, amountOutAfterFees, _receiver); //@audit-issue CEI pattern is not followed
    _decreasePoolAmount(_collateralToken, usdToTokenMin(_collateralToken, usdOut));      You, 22 minutes ago • Un
    return amountOutAfterFees;
}
```

Recommendation

We recommend adhering to the CEI pattern instead.

Resolution

Fixed

BU-1 | Contract is susceptible to a phantom address attack

Severity	Category	Status
● LOW	Phantom addresses	Acknowledged

Description of the issue

The Balance Updater contract allows to update the balance of specific vaults with specific tokens that have a whitelist validation on the vault:

```
function updateBalance(
    address _vault,
    address _token,
    address _usdg,
    uint256 _usdgAmount
) public {
    IVault vault = IVault(_vault);
    IERC20 token = IERC20(_token);
    uint256 poolAmount = vault.poolAmounts(_token); //0
    uint256 fee = vault.feeReserves(_token);  //0
    uint256 balance = token.balanceOf(_vault);

    uint256 transferAmount = poolAmount.add(fee).sub(balance);
    token.transferFrom(msg.sender, _vault, transferAmount);
    IERC20(_usdg).transferFrom(msg.sender, _vault, _usdgAmount);
    vault.sellUSDG(_token, msg.sender);
}
```

Users can pass malicious addresses as _vault and _token to try to somehow find a way of not sending a token or interacting with the wrong vault. We found no way to make this profitable in any way, but there should at least be validation on the _vault address so they are in fact vaults from Lexer.

Recommendation

Validate that the _vault parameter is in fact a vault from lexer and not a malicious contract with the IVault interface. Having a mapping storing the vault/vaults that lexer uses should solve the issue.

Resolution

Acknowledged. The contract will not be used but there are no changes on the smart contract level

BS-1 | Lenght of both arrays is not compared allowing a mismatch between lengths to happen.

Severity	Category	Status
● LOW	Incorrect Validation	Resolved

Description of the issue

The Batch Sender contract allows to send multiple amounts of a certain token to multiple addresses at the same time:

```
  function _send(IERC20 _token, address[] memory _accounts, uint256[] memory
_amounts, uint256 _typeId) private {
        for (uint256 i = 0; i < _accounts.length; i++) {
            address account = _accounts[i];
            uint256 amount = _amounts[i];
            _token.transferFrom(msg.sender, account, amount);
        }

        emit BatchSend(_typeId, address(_token), _accounts, _amounts);
    }
```

Currently, there is no check that the array of addresses and the array of amount have the same length allowing for mistakes to happen and possibly to have mistaken amounts sent to the wrong addresses.

Recommendation

Do check that both arrays have the same length, if not revert the execution.

Resolution

Fixed

SXLPM-2 | Cooldown period can be bypassed.

Severity	Category	Status
● LOW	Bypass check	Acknowledged

Description of the issue

On both XLPM and SXLPM contracts, there is a cooldown period that is “activated” every time you add liquidity. Then, this cooldown period is enforced when withdrawing the liquidity so you can’t really arbitrage in the same block.

```
function _removeLiquidity(
        address _account,
        address _tokenOut,
        uint256 _xlpAmount,
        uint256 _minOut,
        address _receiver
    ) private returns (uint256) {
        require(_xlpAmount > 0, "XlpManager: invalid _xlpAmount");
        require(
            lastAddedAt[_account].add(cooldownDuration) <= block.timestamp,
            "XlpManager: cooldown duration not yet passed"
        );
```

This check can be bypassed by creating 2 contracts. The first one will be the depositor, and the second one will be the withdrawer.
First contract will add liquidity and receive XLP tokens in exchange. This tokens will then be transferred to the withdrawer contract bypassing the msg.sender verification, and the withdrawer contract will be able to withdraw the tokens.

Note there has not been found a profitable way of exploiting the issue, but we are still raising the attack vector in case there is something we missed.

Recommendation

Do not allow a user to supply and withdraw in the same block.

Resolution

Acknowledged issue. The team will document it.

SLT-3 | Use safeTransfer instead of transfer

Severity	Category	Status
● LOW	Incorrect Design	Resolved

Description of the issue

There are several instances across the codebase where any token can be used and transfer/transferFrom is being used to perform the transfer.

```
IERC20(_token).transferFrom(_sender, address(this), _amount);
```

Recommendation

Consider integrating with OZ library to use safeTransfer instead.

Resolution

Fixed

GLOBAL-3 | Missing events in functions that update key state variables.

Severity	Category	Status
● LOW	Incorrect Design	Acknowledged

Description of the issue

Across the whole codebase there are several functions that do change ownership and permissions, or they update key states of the codebase, but they miss events.

Some example of functions missing events:

```
setInfo()
setHandler()
setInPrivateTransferMode()
withdrawToken()
removeAdmin()
addAdmin()
setYieldTrackers()
setGov()
claim()
recoverClaim()
removeNonStakingAccount()
addNonStakingAccount()
```

Recommendation

Add events for each function that needs them and emit them accordingly.

Resolution

Acknowledged

GLOBAL-4 | Lack Of Input Validation

Severity	Category	Status
● LOW	Input Validation	Resolved

Description of the issue

Throughout the codebase there is a lack of input validation

Recommendation

Consider adding input validation on all crucial parameters that are being set

Resolution

Fixed

GLOBAL-5 | Keepers can be griefed in any function

Severity	Category	Status
● LOW	Griefing	Acknowledged

Description of the issue

In contracts where keepers have to relay transactions, such as Orderbook, Position Manager, Position Router, and its Synthetic versions, any transaction a keeper performs, such as executing increase/decrease orders or canceling orders, can be grieved by the user who created the order by front-running the keeper. Because Lexer is deploying primarily in Arbitrum, the severity is not as high as in other chains.

Recommendation

Be aware of such potential risks.

Resolution

Acknowledged

GLOBAL-6 | Transfer-tax tokens are not supported

Severity	Category	Status
● LOW	Incorrect Design	Acknowledged

Description of the issue

The whole architecture has several spots that will not work with transfer-tax tokens. This will break the entire system.

Recommendation

Consider not using such tokens

Resolution

Acknowledged

GLOBAL-7 | Using safeMath in Solidity 0.8

Severity	Category	Status
● LOW	Incorrect Design	Acknowledged

Description of the issue

Lexer refactored GMX's v1 codebase from version 0.6 to 0.8. The library safeMath is made to account for under/overflows, which starting from version 0.8.0 have already been accounted for in the EVM. Therefore, the use of safeMath is redundant.

Recommendation

As this is a non-trivial issue and it is found across the whole codebase, refactoring to not using safeMath would be prone to making mistakes in a lot of places. For such a large codebase, the best would probably be to leave it as it is.

Resolution

Acknowledged.

SXLPM-3 | GetPrice should be use very carefully externally

Severity	Category	Status
● INF	Informational	Resolved

Description of the issue

Currently, from SXLPManager and XLPManager you can fetch the price either maximized or not. This terminology of maximizing prize will surely be confusing for anyone integrating with lexer. The idea is that the user should always be given less so they can't profit from arbitraging.

```
function getPrice(bool _maximise) external view returns (uint256) {
      uint256 aum = getAum(_maximise);
      uint256 supply = IERC20(xlp).totalSupply();
      return aum.mul(XLP_PRECISION).div(supply);
  }
```

Recommendation

The cases for when to maximise or not to maximise the price should be added in a code comment or/and in the docs, so integrations calling this function do not make critical mistakes.

Resolution

Fixed. Users are going to be provided with documentation about how to use the function and where to use `_maximize`

SXLPM-4 | Several variables should be named xlp instead of glp

Severity	Category	Status
● INF	TYPO	Resolved

Description of the issue

Lexer has re-named the original GLP token from GMX to XLP, though there are several spots in both the synthetic manager and the non-synthetic manager where they still reference its name as glp.

```
   function addLiquidity(
        address _token,
        uint256 _amount,
        uint256 _minUsdg,
        uint256 _minGlp, //@audit-issue INF this should be named _minXlp not
_minGlp
        bytes[] calldata _priceUpdateData
     ) external payable nonReentrant returns (uint256) {
```

Recommendation

Re-name those variables from _minGlp to _minXlp

Resolution

Fixed

GLOBAL-8 | Current architecture allows to back-run transfers if certain scenarios are met.

Severity	Category	Status
● INF	Architectural Error	Partially Resolved

Description of the issue

The current architecture of the codebase tracks the funds sent to the vault with a transferin function that does check the balance of the contract post transfer.

```
function _transferIn(address _token) private returns (uint256) {
       uint256 prevBalance = tokenBalances[_token];
       uint256 nextBalance = IERC20(_token).balanceOf(address(this));
       tokenBalances[_token] = nextBalance;

       return nextBalance - (prevBalance);
   }
```

This is fine as it is if you interact with the vault from their manager contract that directly forward the funds with a transfer of the tokens in the same transaction.
In case that the `inManagerMode` is turned off:

```
function _validateManager() private view {
       if (inManagerMode) {
           _validate(isManager[msg.sender], 54);
       }
   }
```

users will be able to interact directly with the vault, which means that either they should transfer the funds in a different transaction or create their own contract to transfer the funds and directly call the respective function.
If the first option is chosen, an attacker can back-run them and use the funds of the user to their own benefit.

Recommendation

Explicitly document that funds should never be transferred in different transactions and consider not disabling the `inManagerMode` because it opens a lot of new attack vectors.

Resolution

No changes on the smart contracts, but `inManagerMode` will never be disabled.

DISCLAIMER

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Marc Weiss and 0xKato to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Blockchain technology and cryptographic assets present a high level of ongoing risk.

Our position is that each company and individual are responsible for their own due diligence and continuous security. Our goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze. Therefore, we do not guarantee the explicit security of the audited smart contract, regardless of the verdict.