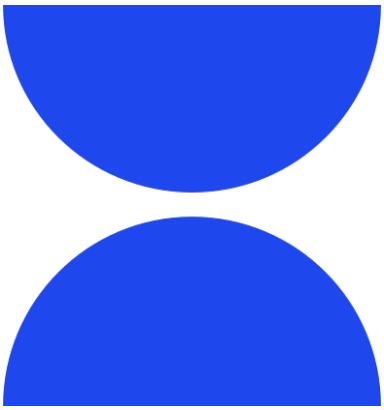
verichains

SECURITY AUDIT OF

BSX SMART CONTRACT



Public Report

Jul 11, 2025

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jul 11, 2025. We would like to thank the BSX Exchange for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the BSX Smart Contract. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code. BSX Exchange fixed or acknowledged the issues.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About BSX Smart Contract

BSX, a self-custodial crypto derivatives decentralized exchange (DEX) built on the Base Layer 2 (L2) network. BSX features a non-custodial system of "off-chain execution and on-chain settlement" that merges the security and transparency of a decentralized exchange with the experience and speed of a centralized exchange. Its architecture design consists of an orderbook model that works with various market makers to offer extensive and liquid access to different assets, instant execution, and high throughput via BSX's comprehensive technical stack.

1.2. Audit Scope

This audit focused on identifying security flaws in code and the design of BSX Smart Contract.

It was conducted on commit `e428fb59d09f7784b1a2617291386cf84028a3ee` from the GitHub repository *bsx-exchange/contracts-core*.

SHA256 Sum	File
c3e79d7dcc8811e19946e61c58c1b241f717b0dfc3613ca2dcddfa0aee759368	contracts/1000x/BSX1000x.sol
4ebea34567f167260209a00f14a0d42959456fa6fa74da14da57d1332ead281e	contracts/1000x/interfaces/IBSX1000x.sol
8072ef0372299d3c7546e41bd8bf70170edb6c554a7190e72d9bb4381aafc40a	contracts/exchange/access/Access.sol
c9a07ec27d621e560dd8c0361396f0b80dc2e5a268c1dca6152988d5cd1a247e	contracts/exchange/ClearingService.sol
a3d3e6f0471c40e8381e845a08392416691d0db15bcf81719518c790081d2657	contracts/exchange/Exchange.sol
800d392e3d9e65360f4f5f38a21f88a19b9e515b82c18226c2c227f5a878eed5	contracts/exchange/ExchangeStorage.sol

e970bb7cd5be0aba9e94e8e7e8891a853c08f1c33cb890a3f09a430099724b46	contracts/exchange/interfaces/external/IERC20Extend.sol
10efd1554f37b518a87ad9b35d34e4a23efe58eb6e63394a1aeb7014481a2e98	contracts/exchange/interfaces/external/IERC3009Minimal.sol
172368deff0f8dc8be5dcfd1db307720f75825485961b887fcdef52a3d10132b	contracts/exchange/interfaces/external/IUniversalRouter.sol
021931cecca70a3c3014b468635cc570f08a16b0550b2cbfa61a784cba0e1447	contracts/exchange/interfaces/external/IUniversalSigValidator.sol
8169bfcbe1a2a90af7f2d8fceab7b8179f896a361f9e301c55e7e10b607da0fd	contracts/exchange/interfaces/external/IWETH9.sol
2f37ea05199e04a01469602abdd66efa40f18e71fb58d875159c1699ba75356e	contracts/exchange/interfaces/IClearingService.sol
5a6f0a0b3da38ce9928b9734d6203642e904d5739932b541c6306091102cc3ab	contracts/exchange/interfaces/IExchange.sol
ed60583c63eabb889568f39973585757efc8d31f9b0027cc5910e0e214439b5e	contracts/exchange/interfaces/ILiquidation.sol
1fc747fe88f5fb61007ffa83aee030a754a84f585678532e7e86014d2e9c053d	contracts/exchange/interfaces/IOrderBook.sol
4e9902c5c6a8ad99d2340e432f15d8172d7d79b2b00fb98d1dbd3a52518ac5b1	contracts/exchange/interfaces/IPerp.sol
6e749a0be6a62debdb666741b139df4e98a4a52b8005e4cb9e559d3aaebe5e9a	contracts/exchange/interfaces/ISpot.sol
48a4326873da0c2430d330338f5b1cc7553f353ed644416c6c5e18a407bf68dc	contracts/exchange/interfaces/ISwap.sol

15ed15491b03aec133dce707ec2531e4cdcf02f53b6b5f3bf9624984fe8641c5	contracts/exchange/interfaces/IVaultManager.sol
ee49f70df92d2fd5a208fd247067d683eb9a477580bf3901fad954482887b7c5	contracts/exchange/lib/Commands.sol
c325e9738e16674c80e264d98cec231a0e7a4661657cb22f5d1c931e0c73c7b8	contracts/exchange/lib/Errors.sol
3e4c6688f2ebd217616c7a8b08b3c3f66f64b88b9c87ec5177cee2dd1e6e6b1e	contracts/exchange/lib/logic/AccountLogic.sol
e52e74524ba51683ca8abb74493927979774f48970febc0098d44a4bcb343221	contracts/exchange/lib/logic/AdminLogic.sol
d84903808ea0e995d8c36bb111b7100bdbf2bbfb5affedbb4fe37b844cba453f	contracts/exchange/lib/logic/BalanceLogic.sol
280b085208dcac4a183ac35f0462b4178fb6c01953837ebf937ad7f6c5804daf	contracts/exchange/lib/logic/GenericLogic.sol
953d3ded8233b10b8c9140287036d1d19ff035eeb8025e7fa35dac12a25765d9	contracts/exchange/lib/logic/LiquidationLogic.sol
7671f267b1a5967888a26125c52f84eee6e72ba5e0cda874568db41f218b4f87	contracts/exchange/lib/logic/OrderLogic.sol
792c492b9cde6f4c7d7b73da65c6368972621a89510668b0b4a389694cf20049	contracts/exchange/lib/logic/SignerLogic.sol
ba8c439055f1565014beb30cf11e1f8655972fc0939df3aaa4714be57ff7389e	contracts/exchange/lib/logic/SwapLogic.sol
166cab7f74bc3821c823cbedbeb2137a6dcad769a20389ce23230280c6ef3988	contracts/exchange/lib/MathHelper.sol

19b5cb1fe7f98e2c3fbecac1609f3e0398a1f3d49d417d29cf15eab8b40aa042	contracts/exchange/lib/Percentage.sol
06773ad2ca05571a8ed1d6b5cbf074f182df6ce527382808e8b41a6c286e1bd9	contracts/exchange/lib/Roles.sol
23769f9bc288da53acf0abd31f07638301c90d228a4445fc6242f723157715bf	contracts/exchange/OrderBook.sol
cbcefc9e16e891f6866f36c208192f26e494c6bc11bb47da03f6991d38a1e785	contracts/exchange/Perp.sol
8a3d4e04aed72fc41e8a8d935bac2b44d7f90160dd54d529f01f85f81d16b9ba	contracts/exchange/share/Constants.sol
7d6824235cce2ed4810d2a03f92bfe5debc544e6935d8542ffad34ac4fc86879	contracts/exchange/share/Enums.sol
414134c789e1b869a698194d78a5f7b079f2ce74714b26828b88550b45a6a113	contracts/exchange/Spot.sol
19f1a88e7111ff3305ee27e428c1a0f7ea60f529dfcc658121e8dc5b22a9a885	contracts/exchange/VaultManager.sol
7d90c94dc479b5e245eee6f04589b93eef2dd920d2a06e5a2e3c9f0e1d28aad0	contracts/external/chainlink/IChainlinkAggregatorV3.sol
543e79b6d6b8ba098a640398b1443d1bc8a4980cf07d8acf27151c284ec13b6e	contracts/misc/BsxOracle.sol
de6bfb789fc045d5369128eee1c10c8d04df64209cdd9745be908bd193653f54	contracts/misc/interfaces/IBsxOracle.sol

Table 1. Audit Scope

1.3. Audit Methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 2. Severity levels

1.4. Disclaimer

BSX Exchange acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. BSX Exchange understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, BSX Exchange agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the BSX Exchange will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the BSX Exchange, the final report will be considered fully accepted by the BSX Exchange without the signature.

2. AUDIT RESULT

2.1. Overview

The BSX protocol smart contracts are written in Solidity version `^0.8.23`. The contracts inherit from OpenZeppelin's libraries and implement upgradeable patterns to allow for future improvements while maintaining security.

Core Exchange

The BSX protocol implements a sophisticated hybrid orderbook system through six interconnected smart contracts. At its core, `Exchange.sol` serves as the main entry point, orchestrating interactions between all components while enforcing role-based access control and enabling atomic batch processing of multiple operations. The `OrderBook.sol` contract powers the hybrid orderbook model by combining off-chain order matching with on-chain settlement - orders are matched quickly off-chain by sequencers but settled securely on-chain with rigorous price and size validation.

- **Exchange.sol**: main entry point
- **OrderBook.sol**: order matching engine
- **Perp.sol**: perpetual futures logic
- **Spot.sol**: spot balance management
- **ClearingService.sol**: settlement & liquidation
- **VaultManager.sol**: yield asset management

BSX1000x Contract

The **BSX1000x contract** is a specialized product within the BSX protocol that enables users to trade with extremely high leverage—up to 1000x—on select assets. Unlike the main exchange, BSX1000x operates without an orderbook. Instead, users can open and close positions directly at the current oracle price, which is submitted from trusted BSX Operator.

- **Up to 1000x leverage:** Users can amplify their exposure significantly with a small amount of collateral.
- **No orderbook:** Trades are executed instantly at the oracle price, simplifying the user experience and reducing latency.

The users can transfer between BSX1000x and the main exchange.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of BSX Smart Contract.

#	Title	Severity	Status
1	Lack of Validation of Fetched Oracle Data	HIGH	FIXED
2	Missing On-Chain Balance Validation	HIGH	ACKNOWLEDGED
3	Missing Leverage Validation	HIGH	ACKNOWLEDGED
4	Missing Spot Balance Lock	HIGH	ACKNOWLEDGED
5	Missing Re-entrancy Protection	MEDIUM	FIXED
6	Incorrect Account Allowance Checking	MEDIUM	FIXED
7	BSX1000x Tolerance Exploitation	MEDIUM	ACKNOWLEDGED
8	Centralization Risks	MEDIUM	ACKNOWLEDGED
9	100% LTV Liquidation Threshold Risk	MEDIUM	ACKNOWLEDGED
10	Missing Hash Nonce in Authorization	LOW	ACKNOWLEDGED
11	ETH Stuck in Contract	LOW	FIXED
12	Supported Token Removal Risk	INFORMATIVE	ACKNOWLEDGED

Table 3. Findings List

2.2.1. HIGH - Lack of Validation of Fetched Oracle Data

Location: `contracts/misc/BsxOracle.sol#L45-50`

The `BsxOracle` contract calls the `latestRoundData` function from ChainLink price feeds. However, there is no check on the return values to validate stale data prices. This could lead to stale prices

```
function getTokenPriceInUsd(address token) external view override returns (uint256) {
    IChainlinkAggregatorV3 aggregator = IChainlinkAggregatorV3(aggregators[token]);
    (, int256 price,,,) = aggregator.latestRoundData();
    uint8 decimals = aggregator.decimals();
    return _scalePrice(price.toUint256(), decimals);
}
```

RECOMMENDATION

Add validation for price staleness, zero/negative values, and acceptable thresholds for latest updates time.

UPDATES

- ***Jul 11, 2025***: The BSX Exchange team has acknowledged the issue and fixed it on commit `f446977`

2.2.2. HIGH - Missing On-Chain Balance Validation

Location:

- contracts/exchange/lib/logic/OrderLogic.sol
- contracts/exchange/OrderBook.sol

The OrderBook contract contains a critical security vulnerability where it executes trades without validating users' on-chain collateral balances or margin requirements. This design flaw allows users to open positions with insufficient funds, potentially creating unlimited leverage and accumulating bad debt that threatens the protocol's solvency.

The matchOrders function processes trades based solely on order parameters without checking if users have adequate collateral:

```
function matchOrders(
    uint8 productIndex,
    Order calldata maker,
    Order calldata taker,
    Fees calldata fees,
    bool isLiquidation
) external onlySequencer {
    //No validation - Executes trades without checking user balances or margin requirements

    //...
    uint128 fillAmount = MathHelper.min(maker.size - filled[maker.orderHash], taker.size -
filled[taker.orderHash]);
    //...
    // Calculate position deltas
    if (taker.orderSide == OrderSide.SELL) {
        takerDelta.productAmount = -fillAmount.safeInt128();
        makerDelta.productAmount = fillAmount.safeInt128();
        takerDelta.quoteAmount = price.mul18D(fillAmount).safeInt128();
        makerDelta.quoteAmount = -takerDelta.quoteAmount;
    }
    //...
    // Process trade without validating user has sufficient margin
    _settleBalance(productIndex, maker.sender, makerDelta.productAmount,
makerDelta.quoteAmount, price);
    _settleBalance(productIndex, taker.sender, takerDelta.productAmount,
takerDelta.quoteAmount, price);
    //...
}
function _settleBalance(uint8 _productIndex, address _account, int128 _matchSize, int128
_quote, uint128 _price)
    internal
    returns (int128, int128)
{
    // ... funding calculations ...

    int128 amountToSettle;
```

```
    if (balance.size.mul18D(newSize) < 0) {
        amountToSettle = newQuote + newSize.mul18D(_price.safeInt128());
    } else if (newSize == 0) {
        amountToSettle = newQuote;
    }

    //No validation - Updates balance regardless of available collateral
    spotEngine.updateBalance(_account, _collateralToken, amountToSettle);

    // ...
    return (newQuote, newSize);
}
```

RECOMMENDATION

Implement comprehensive on-chain balance and margin validation before executing any trades.

Sample code to fix the issue:

```
function matchOrders(
    uint8 productIndex,
    Order calldata maker,
    Order calldata taker,
    Fees calldata fees,
    bool isLiquidation
) external onlySequencer {
    // Add pre-trade validation
    _validateMarginRequirements(maker, makerDelta, fees.maker, fees.isMakerFeeInBSX);
    _validateMarginRequirements(taker, takerDelta, fees.taker, fees.isTakerFeeInBSX);

    // Existing trade logic...

    // Add post-trade solvency check
    _ensureUserSolvency(maker.sender);
    _ensureUserSolvency(taker.sender);
}

function _validateMarginRequirements(
    Order memory order,
    Delta memory delta,
    int128 tradingFee,
    bool isFeeInBSX
) internal view {
    address user = order.sender;

    // Check current collateral
    int256 currentCollateral = spotEngine.getBalance(_collateralToken, user);
    require(currentCollateral > 0, "Insufficient collateral");
```

```
    // Calculate required margin for new position
    uint256 requiredMargin = _calculateRequiredMargin(user, delta);
    require(currentCollateral >= requiredMargin.safeInt256(), "Insufficient margin");

    // Validate fee payment capability
    if (isFeeInBSX) {
        int256 bsxBalance = spotEngine.getBalance(BSX_TOKEN, user);
        require(bsxBalance >= tradingFee, "Insufficient BSX for fees");
    }
}

function _ensureUserSolvency(address user) internal view {
    // Calculate total margin requirement across all positions
    // Ensure user remains above maintenance margin threshold
    require(_isUserSolvent(user), "Trade would make user insolvent");
}

// Add necessary helper functions for margin calculations
// ...
```

UPDATES

- ***Jul 11, 2025***: The BSX Exchange team acknowledged the issue and clarified that BSX is a hybrid exchange. Core logic is performed off-chain, which is an intentional design decision.

2.2.3. HIGH - Missing Leverage Validation

Location:

- `contracts/exchange/OrderBook.sol`

The `OrderBook` contract fails to enforce the protocol's intended leverage limits (50x) during trade execution. While the protocol is designed to allow maximum 50x leverage, the contract does not validate position sizes against collateral amounts, enabling users to open positions with unlimited leverage that far exceeds the protocol's risk parameters.

The `matchOrders` function calculates and executes trades without validating that the resulting position stays within leverage limits:

```
function matchOrders(
    uint8 productIndex,
    Order calldata maker,
    Order calldata taker,
    Fees calldata fees,
    bool isLiquidation
) external onlySequencer {
    // ...
```

```
    if (taker.orderSide == OrderSide.SELL) {
        takerDelta.productAmount = -fillAmount.safeInt128();
        makerDelta.productAmount = fillAmount.safeInt128();
        takerDelta.quoteAmount = price.mul18D(fillAmount).safeInt128();
        makerDelta.quoteAmount = -takerDelta.quoteAmount;
    }

    //No leverage validation - Executes trades without checking if resulting position
exceeds 50x leverage
    // Position is created regardless of leverage ratio

    //...
    _settleBalance(productIndex, maker.sender, makerDelta.productAmount,
makerDelta.quoteAmount, price);
    _settleBalance(productIndex, taker.sender, takerDelta.productAmount,
takerDelta.quoteAmount, price);
    //...
}
```

RECOMMENDATION

Implement checking to ensure that the leverage ratio of the position does not exceed the protocol's maximum leverage limit (50x). This should be done before executing any trades.

UPDATES

- ***Jul 11, 2025***: The BSX Exchange team acknowledged the issue and clarified that BSX is a hybrid exchange. Core logic is performed off-chain, which is an intentional design decision.

2.2.4. HIGH - Missing Spot Balance Lock

Location:

- `contracts/exchange/OrderBook.sol`

The `OrderBook` contract does not lock or reserve spot balance when opening positions, relying entirely on off-chain validation. This creates a critical window where users can spend their collateral through other operations while having open positions that depend on that same collateral, leading to under-collateralized positions and potential protocol insolvency.

When a position is opened, the contract calculates deltas and updates positions without locking the required collateral. The `_settleBalance` function only updates balances for realized P&L but doesn't lock collateral for open positions.

```
function matchOrders(...) external onlySequencer {
    //...
```

```
    _settleBalance(productIndex, maker.sender, makerDelta.productAmount,
makerDelta.quoteAmount, price);
    //...
}

function _settleBalance(...) internal returns (int128, int128) {
    // Only settles when closing/flipping positions
    if (balance.size.mul18D(newSize) < 0) {
    amountToSettle = newQuote + newSize.mul18D(_price.safeInt128());
    } else if (newSize == 0) {
    amountToSettle = newQuote;
    }
    //No locking - Updates balance without locking collateral

    spotEngine.updateBalance(_account, _collateralToken, amountToSettle);

    //...
    return (newQuote, newSize);
}
```

RECOMMENDATION

Implement a locking mechanism to reserve collateral when positions are opened. This should prevent users from spending their collateral while positions are open, ensuring that the protocol remains solvent.

UPDATES

- ***Jul 11, 2025***: The BSX Exchange team acknowledged the issue and clarified that BSX is a hybrid exchange. Core logic is performed off-chain, which is an intentional design decision.

2.2.5. MEDIUM - Missing Re-entrancy Protection

Location: `contracts/1000x/BSX1000x.sol`

The BSX1000x contract is missing critical security protection against re-entrancy attacks. Re-entrancy occurs when an external contract calls back into the BSX1000x contract before the first function call has finished executing. This creates a window where an attacker can manipulate the contract's state by calling functions like `deposit()`, `withdraw()`, or `openPosition()` multiple times in a single transaction. Since the contract doesn't track that it's already processing a request, each recursive call sees the same initial state, allowing attackers to potentially drain funds, create unauthorized positions, or manipulate balances before the contract realizes what happened.

The BSX1000x contract lacks re-entrancy protection for its public functions. Unlike its `Exchange.sol` counterpart which implements the `ReentrancyGuardUpgradeable`, the BSX1000x contract does not have any protection against re-entrancy attacks.

```
contract BSX1000x is IBSX1000x, Initializable, EIP712Upgradeable {
    using EnumerableMap for EnumerableMap.UintToUintMap;
    using SafeERC20 for IERC20;
    //...
}
```

RECOMMENDATION

Add the `ReentrancyGuardUpgradeable` from OpenZeppelin and implement the nonReentrant modifier on all external functions that modify state.

UPDATES

- ***Jul 11, 2025***: The BSX Exchange team has acknowledged the issue and fixed it on commit `f5db7df`

2.2.6. MEDIUM - Incorrect Account Allowance Checking

Location: `contracts/1000x/BSX1000x.sol#L133`

In the `depositMaxApproved()` function, the contract checks the allowance of an account without verifying that `msg.sender` is actually authorized to use this allowance.

If `account` is not `msg.sender`, the function will still attempt to transfer tokens based on another user's allowance, which will fail at execution time but waste gas and potentially cause other issues.

```
function depositMaxApproved(address account, address token) public {
    uint256 amount = IERC20(token).allowance(account, address(this));
    depositRaw(account, token, amount);
}

function depositRaw(address recipient, address token, uint128 rawAmount) external payable
nonReentrant {
    uint256 amount = token == NATIVE_ETH ? rawAmount : rawAmount.convertToScale(token);
    _deposit(recipient, token, amount, false);
}

function deposit(address account, uint256 amount) public {
    //...

    collateralToken.safeTransferFrom(msg.sender, address(this), rawAmount);

    //...
}
```

RECOMMENDATION

Modify the function to ensure that only the token owner can use their allowance.

UPDATES

- ***Jul 11, 2025***: The BSX Exchange team has acknowledged the issue and fixed it on commit `bd2dd01`

2.2.7. MEDIUM - BSX1000x Tolerance Exploitation

Location: contracts/1000x/BSX1000x.sol#L531-534

The BSX1000x smart contract has an insufficient profit and loss (PnL) validation mechanism, allowing operators to manipulate reported PnL values within a generous error margin.

In the _closePosition function, the contract validates the reported PnL against an expected calculation:

```
function closePosition(
    uint32 productId,
    address account,
    uint256 nonce,
    uint128 closePrice,
    int256 pnl,
    int256 fee,
    bytes memory signature
) external onlyRole(Roles.BSX1000_OPERATOR_ROLE) {
    bytes32 closeOrderHash =
        _hashTypedDataV4(keccak256(abi.encode(CLOSE_POSITION_TYPEHASH, productId, account,
nonce)));
    _validateAuthorization(account, closeOrderHash, signature);

    _closePosition(productId, account, nonce, closePrice, pnl, fee,
ClosePositionReason.Normal);
}

function _closePosition(
    uint32 productId,
    address account,
    uint256 nonce,
    uint128 closePrice,
    int256 pnl,
    int256 fee,
    ClosePositionReason reason
) internal {
    // ...

    position.closePrice = closePrice;

    int256 expectedPnl = position.size.mul18D(closePrice.safeInt128() -
position.openPrice.safeInt128());
    if (pnl > expectedPnl + MARGIN_ERR || pnl < expectedPnl - MARGIN_ERR) {
        revert InvalidPnl();
    }

    // ...
}
```

The issue is that `MARGIN_ERR` is set too high (1e16) with only fixed absolute validation and no percentage-based checks or sanity limits, allowing malicious operators to systematically exploit significant monetary deviations, especially for large positions.

A malicious BSX1000x operator could consistently report PnL values at the acceptable limit, gradually extracting value through systematic small deviations, potentially favoring specific traders or their own positions, with large positions (>$100k) presenting significant financial exploitation opportunities—directly undermining the protocol's core financial integrity.

RECOMMENDATION

Reduce the `MARGIN_ERR` to a more reasonable value, ideally a percentage of the expected PnL, to prevent exploitation. Implement additional checks to ensure that reported PnL values are within a reasonable range relative to the position size and market conditions.

UPDATES

- ***Jul 11, 2025***: The BSX Exchange team has acknowledged the issue and won't fix it due to the design of the protocol.

2.2.8. MEDIUM - Centralization Risks

Location:

- Multiple contracts across the BSX ecosystem

The BSX Protocol contains multiple centralized operator roles with excessive privileges that can manipulate protocol parameters, bypass security checks, and potentially drain user funds. These centralization risks create single points of failure and expose users to malicious or compromised operator actions.

RECOMMENDATION

- Implement a multi-signature (multi-sig) mechanism for critical administrative functions to distribute control and reduce the risk of a single point of failure.
- Introduce time-locks for sensitive operations to provide users with a window to react to potentially malicious changes.
- Use role-based access control (RBAC) with granular permissions to limit the scope of operator privileges.
- Regularly audit and monitor operator actions to ensure accountability and transparency.
- Decentralize governance by involving the community in decision-making processes through voting mechanisms or decentralized autonomous organization (DAO) structures.

UPDATES

- ***Jul 11, 2025***: The BSX Exchange team has acknowledged the issue.

2.2.9. MEDIUM - 100% LTV Liquidation Threshold Risk

Location:

- contracts/exchange/OrderBook.sol

The BSX Protocol allows liquidations at 100% Loan-to-Value (LTV) ratio, meaning positions can be liquidated when the collateral value exactly equals the position value. However, this creates a critical flaw where liquidations are guaranteed to result in negative balances and bad debt due to real-time price movements, liquidation fees, trading fees, and funding payments that occur between liquidation trigger calculation and on-chain execution.

RECOMMENDATION

Implement smaller LTV liquidation threshold with dynamic fee buffers and partial liquidation capabilities to ensure protocol sustainability during volatile market conditions.

UPDATES

- ***Jul 11, 2025***: The BSX Exchange team has acknowledged the issue and won't fix it due to the design of the protocol.

2.2.10. LOW - Missing Hash Nonce in Authorization

Location: contracts/exchange/lib/logic/AccountLogic.sol#L80-86

The subaccount creation system has a security flaw that allows signature replay attacks. When users want to create a subaccount, they must provide digital signatures to prove authorization. However, the system doesn't include a "nonce" (a unique, one-time-use number) in the data being signed. This means that once a valid signature is created and used, an attacker who captures that signature can replay it indefinitely to create unauthorized subaccounts. Think of it like a signed check without a check number - once someone gets a copy of your signed check, they can photocopy it and use it multiple times. In the blockchain context, all transactions are public, so anyone can see and copy the signature data from successful subaccount creation transactions. An attacker could then use these copied signatures to create additional subaccounts without the user's knowledge or consent, potentially gaining unauthorized access to user funds or positions.

```
function createSubaccount(
    mapping(address => IExchange.Account) storage accounts,
    Access access,
    IExchange exchange,
    address main,
    address subaccount,
    bytes memory mainSignature,
    bytes memory subSignature
```

```
) external {
//...
    bytes32 hash =
exchange.hashTypedDataV4(keccak256(abi.encode(CREATE_SUBACCOUNT_TYPEHASH, main,
subaccount)));
    if (!UNIVERSAL_SIG_VALIDATOR.isValidSig(main, hash, mainSignature)) {
        revert Errors.Exchange_InvalidSignature(main);
    }
    if (!UNIVERSAL_SIG_VALIDATOR.isValidSig(subaccount, hash, subSignature)) {
        revert Errors.Exchange_InvalidSignature(subaccount);
    }
//...
}
```

RECOMMENDATION

Include a nonce in all signature verification operations and maintain state to prevent reuse.

UPDATES

- ***Jul 11, 2025***: The BSX Exchange team has acknowledged the issue.

2.2.11. LOW - ETH Stuck in Contract

Location: `contracts/exchange/lib/logic/BalanceLogic.sol:54-59`

In the `deposit()` function, when handling non-native deposits, if user mistakely sends ETH when calling deposit, the contract does not check `msg.value` and will not revert, leading to ETH being stuck in the contract.

```
function deposit(
    mapping(address => IExchange.Account) storage accounts,
    BalanceEngine calldata engine,
    address recipient,
    address token,
    uint256 amount,
    bool earn
) external {
    //...

    if (token == NATIVE_ETH) {
        //...
    } else {
        (uint256 roundDownAmount, uint256 rawAmount) =
amount.roundDownAndConvertFromScale(token);
        if (roundDownAmount == 0 || rawAmount == 0) revert Errors.Exchange_ZeroAmount();
        amount = roundDownAmount.safeUInt128();
        IERC20(token).safeTransferFrom(msg.sender, address(this), rawAmount);
    }
```

```
    //...
}
```

RECOMMENDATION

Check `msg.value` in the `deposit()` function and revert if it is not zero when the token is not `NATIVE_ETH`. This will prevent ETH from being stuck in the contract.

UPDATES

- ***Jul 11, 2025***: The BSX Exchange team has acknowledged the issue and fixed it on commit `e428fb5`

2.2.12. INFORMATIVE - Supported Token Removal Risk

Location: `contracts/exchange/Exchange.sol#removeSupportedToken()`

Tokens can be removed from supported list without checking if users have balances, potentially stranding funds.

```
function removeSupportedToken(address token) external onlyRole(Roles.GENERAL_ROLE) {
    return AdminLogic.removeSupportedToken(supportedTokens, token);
}
```

RECOMMENDATION

Add a check to ensure that no user has a non-zero balance of the token before allowing its removal. This will prevent users from being unable to access their funds if a token is removed while they still hold it.

UPDATES

- ***Jul 11, 2025***: The BSX Exchange team has acknowledged the issue.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Jul 11, 2025*	Public Report	Verichains Lab

Table 4. Report versions history