EXVUL

SMART CONTRACT AUDIT REPORT

AntX Smart Contract

JANUARY 2026

Contents

1. EXECUTIVE SUMMARY

ExVul Web3 Security was engaged by **AntX** to review smart contract implementation. The assessment was conducted in accordance with our systematic approach to evaluate potential security issues based upon customer requirement. The report provides detailed recommendations to resolve the issue and provide additional suggestions or recommendations for improvement.

The outcome of the assessment outlined in chapter 3 provides the system's owners a full description of the vulnerabilities identified, the associated risk rating for each vulnerability, and detailed recommendations that will resolve the underlying technical issue.

1.1 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10] which is the gold standard in risk assessment using the following risk models:

- **Likelihood**: represents how likely a particular vulnerability is to be uncovered and exploited in the wild.
- **Impact**: measures the technical loss and business damage of a successful attack.
- **Severity**: determine the overall criticality of the risk.

Likelihood can be: High, Medium and Low and impact are categorized into: High, Medium, Low, Informational. Severity is determined by likelihood and impact and can be classified into five categories accordingly: Critical, High, Medium, Low, Informational shown in table 1.1.

Likelihood \ IMPACT	Informational	Low	Medium	High
High	INFO	MEDIUM	HIGH	CRITICAL
Medium	INFO	LOW	MEDIUM	HIGH
Low	INFO	LOW	LOW	MEDIUM

Table 1.1 Overall Risk Severity

To evaluate the risk, we will be going through a list of items, and each would be labelled with a severity category. The audit was performed with a systematic approach guided by a comprehensive assessment list carefully designed to identify known and impactful security issues. If our tool or analysis does not identify any issue, the contract can be considered safe regarding the assessed item. For any discovered issue, we might further deploy contracts on our private test environment and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.2.

- **Basic Coding Bugs**: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- **Code and business security testing**: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- **Additional Recommendations**: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Category	Assessment Item
Basic Coding Assessment	• Apply Verification Control • Authorization Access Control • Forged Transfer Vulnerability • Forged Transfer Notification • Numeric Overflow • Transaction Rollback Attack • Transaction Block Stuffing Attack • Soft Fail Attack • Hard Fail Attack • Abnormal Memo • Abnormal Resource Consumption • Secure Random Number

Advanced Source Code Scrutiny	• Asset Security • Cryptography Security • Business Logic Review • Source Code Functional Verification • Account Authorization Control • Sensitive Information Disclosure • Circuit Breaker • Blacklist Control • System API Call Analysis • Contract Deployment Consistency Check • Abnormal Resource Consumption
Additional Recommendations	• Semantic Consistency Checks • Following Other Best Practices

Table 1.2: The Full List of Assessment Items

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [14], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development.

2. FINDINGS OVERVIEW

2.1 Project Info And Contract Address

Project Name	Audit Time	Language
AntX	06/01/2026 - 14/01/2026	Solidity

Official Website

https://www.antxfi.com/

Repository

https://github.com/antxprotocol/contracts/

Commit Hash

034a8a5bfb1629559c3c16c7f8974dd90eac9f74

2.2 Summary

Severity	Found
CRITICAL	1
HIGH	2
MEDIUM	5
LOW	2
INFO	3

2.3 Key Findings

Severity	Findings Title	Status
CRITICAL	Broken Forwarder Semantics in Stargate Adapter Causes Transaction Failure	Fixed
HIGH	batchWithdraw will be subject to a DOS attack due to malicious parameters	Acknowledge
HIGH	Lack of Access Control and Parameter Validation Enables Bypass ChainId	Fixed
MEDIUM	Continuous batch updates can indefinitely delay force withdrawals	Acknowledge
MEDIUM	Emergency withdraw signatures lack one-time use protection	Fixed
MEDIUM	_userWithdraw did not charge a transaction fee	Acknowledge
MEDIUM	Stargate Pool Token Mismatch Enables Fund Loss and Theft	Fixed
MEDIUM	riskTiers Unverified Ranking	Fixed
LOW	Zero-initialized lastBatchTime enables immediate force withdrawals	Fixed
LOW	Sign flip caused by uint256-to-int256 conversion overflow	Fixed
INFO	Redundant batch length checks increase gas costs	Fixed
INFO	Incorrect Error Type Used in Constructor	Fixed
INFO	Antex naming error	Acknowledge

Table 2.3: Key Audit Findings

3. DETAILED DESCRIPTION OF FINDINGS

3.1 Broken Forwarder Semantics in Stargate Adapter Causes Transaction Failure

SEVERITY: CRITICAL **STATUS:** Fixed

PATH:

`AntStrargateAdapter.sol`

DESCRIPTION:

sendToken directly forwards the call to stargate.sendToken. Inside stargate.sendToken, it attempts to transfer tokens from msg.sender to Stargate. However, msg.sender is the Adapter contract, not the original StargateWithdrawcall or User.

```
// AntStrargateAdapter.sol
    function sendToken(SendParam calldata _sendParam, MessagingFee
    calldata _fee, address _refundAddress)
        external
        payable
        returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
    oftReceipt, Ticket memory ticket)
    {
        return stargate.sendToken{value: msg.value}(_sendParam, _fee,
    _refundAddress);
    }
Stargate V2's sendToken logic relies on msg.sender for token deduction.
// https://github.com/stargate-protocol/stargate-v2/blob/
// 173cb957ebafa51f3287f28c8cfcdfaf810b47a7
// packages/stg-evm-v2/src/StargateBase.sol#L247
    function sendToken(
        SendParam calldata _sendParam,
        MessagingFee calldata _fee,
        address _refundAddress
    )
        public
        payable
        override
        nonReentrantAndNotPaused
```

```
        returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
    oftReceipt, Ticket memory ticket)
    {
        // step 1: assets inflows and apply the fee to the input amount
        (bool isTaxi, uint64 amountInSD, uint64 amountOutSD) =
    _inflowAndCharge(_sendParam);
        // ...
    }

    function _inflowAndCharge(
        SendParam calldata _sendParam
    ) internal returns (bool isTaxi, uint64 amountInSD, uint64
    amountOutSD) {
        isTaxi = _isTaxiMode(_sendParam.oftCmd);
        //@audit msg.sender is Adapter but not User
        amountInSD = _inflow(msg.sender, _sendParam.amountLD);
        // ...
    }

// https://github.com/stargate-protocol/stargate-v2/blob/
// 173cb957ebafa51f3287f28c8cfcdfaf810b47a7
// packages/stg-evm-v2/src/StargatePool.sol#L260
    function _inflow(address _from, uint256 _amountLD) internal virtual
    override returns (uint64 amountSD) {
        amountSD = _ld2sd(_amountLD);
        Transfer.safeTransferTokenFrom(token, _from, address(this),
    _sd2ld(amountSD)); // remove the dust and transfer
    }

// https://github.com/stargate-protocol/stargate-v2/blob/
// 173cb957ebafa51f3287f28c8cfcdfaf810b47a7
// packages/stg-evm-v2/src/libs/Transfer.sol#L112
    function safeTransferTokenFrom(address _token, address _from, address
    _to, uint256 _value) internal {
        if (!transferTokenFrom(_token, _from, _to, _value)) revert
    Transfer_TransferFailed();
    }

    function transferTokenFrom(
        address _token,
        address _from,
        address _to,
        uint256 _value
```

```
    ) internal returns (bool success) {
        success = _call(_token,
    abi.encodeWithSelector(IERC20(_token).transferFrom.selector, _from,
    _to, _value));
    }
```

IMPACT:

In Stargate, msg.sender is the Adapter contract, not the original StargateWithdrawor User. Adapter never approved Stargate to spend its funds. Stargate will always fail to transferFrom. Any attempt to use this adapter for token transfers will revert. 100% of transactions fail.

RECOMMENDATIONS:

Approve Stargate then call Stargate.

```
+ import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
+ import {SafeERC20} from "
    @openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

+ contract AntStrargateAdapter is IStargate {
+     using SafeERC20 for IERC20;
// ...
  function send(SendParam calldata _sendParam, MessagingFee calldata _fee,
    address _refundAddress)
      external
      payable
      returns (MessagingReceipt memory receipt, OFTReceipt memory
    oftReceipt)
  {
+     address token = stargate.token();
+     if (token != address(0)) {
+         IERC20(token).safeTransferFrom(msg.sender, address(this),
    _sendParam.amountLD);
+         IERC20(token).forceApprove(address(stargate),
    _sendParam.amountLD);
+     }
-     return stargate.send{value: msg.value}(_sendParam, _fee,
    _refundAddress);
+     (receipt, oftReceipt) = stargate.send{value: msg.value}(_sendParam,
    _fee, _refundAddress);
```

```
+     if (token != address(0)) {
+         IERC20(token).forceApprove(address(stargate), 0);
+     }
+     return (receipt, oftReceipt);
 }

 function sendToken(SendParam calldata _sendParam, MessagingFee calldata
   _fee, address _refundAddress)
     external
     payable
     returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
   oftReceipt, Ticket memory ticket)
 {
+     address token = stargate.token();
+     if (token != address(0)) {
+         IERC20(token).safeTransferFrom(msg.sender, address(this),
   _sendParam.amountLD);
+         IERC20(token).forceApprove(address(stargate),
   _sendParam.amountLD);
+     }
-     return stargate.sendToken{value: msg.value}(_sendParam, _fee,
   _refundAddress);
+     (msgReceipt, oftReceipt, ticket) = stargate.sendToken{value:
   msg.value}(_sendParam, _fee, _refundAddress);
+     if (token != address(0)) {
+         IERC20(token).forceApprove(address(stargate), 0);
+     }
+
+     return (msgReceipt, oftReceipt, ticket);
 }
```

3.2 batchWithdraw will be subject to a DOS attack due to malicious parameters

SEVERITY: HIGH **STATUS:** Acknowledge

PATH:

src/Asset.sol

DESCRIPTION:

batchWithdraw calls _userWithdraw in a loop. Since the _userWithdraw function contains a large number of checks, the entire transaction will be reverted if any of these checks fail.

```
    function _userWithdraw(
        uint256 clientOrderId,
        bytes32 user,
        bytes32 recipient,
        uint256 expireTime,
        uint64 dstChainId,
        uint256 amount,
        uint256 fee,
        bytes memory signatures,
        bool isForce,
        SignatureType signatureType
    ) internal validAmount(amount) {
        if (!isForce) {
            // check if the clientOrderId is already used
            if (usedClientOrderIds[clientOrderId]) revert
    ClientOrderIdAlreadyUsed();
            usedClientOrderIds[clientOrderId] = true;

            // check if the expireTime is expired
            if (expireTime < block.timestamp) revert ExpiredTransaction();

            // check user signature
            bytes32 operationHash =
                _hashUserWithdraw(clientOrderId, user, recipient, amount,
    fee, expireTime, dstChainId);
            operationHash =
    MessageHashUtils.toEthSignedMessageHash(operationHash);
```

```
        if (signatureType == SignatureType.ECDSA) {
            if (user !=
bytes32(uint256(uint160(ECDSA.recover(operationHash, signatures))))) {
                revert InvalidUserSignature();
            }
        } else {
            revert NotSupportedSignatureType();
        }
    }

    uint256 userAvailableAmount = availableAmount(user);
    if (userAvailableAmount < amount) revert
InsufficientUserBalance(userAvailableAmount, amount);

    if (dstChainId == block.chainid) {
        uint256 preBalance = USDC.balanceOf(address(this));

IERC20(USDC).safeTransfer(address(uint160(uint256(recipient))),
amount);

        uint256 postBalance = USDC.balanceOf(address(this));
        assert(preBalance - postBalance == amount);
        // emit event
        emit UserWithdraw(clientOrderId, user, recipient, amount,
dstChainId);
    } else {
        USDC.forceApprove(address(stargateWithdraw), amount);

        (uint256 valueToSend, SendParam memory sendParam,
MessagingFee memory messagingFee) =
            stargateWithdraw.prepareTakeTaxi(dstChainId, amount,
recipient);

        if (address(this).balance < valueToSend) {
            revert InsufficientEthBalance(valueToSend,
address(this).balance);
        }

        stargateWithdraw.crossChainWithdraw{
            value: valueToSend
        }(clientOrderId, user, amount, dstChainId, recipient,
address(this), sendParam, messagingFee);
```

```
            USDC.forceApprove(address(stargateWithdraw), 0);

            emit CrossChainWithdraw(clientOrderId, user, recipient,
    amount, dstChainId);
        }
    }
```

IMPACT:

Attackers can revert batchWithdraw transactions by means of various malicious parameters, thereby causing a DoS on the batchWithdraw function.

RECOMMENDATIONS:

Use try…catch to catch errors in the _userWithdraw function without reverting the transaction.

```
       function batchWithdraw(
        uint256[] memory clientOrderIds,
        uint64[] memory subaccountIds,
        bytes32[] memory recipients,
        uint256[] memory expireTimes,
        uint256[] memory amounts,
        uint256[] memory fees,
        bytes[] memory signatures,
        uint64[] memory dstChainIds,
        SignatureType signatureType
    ) external nonReentrant onlyWithdrawOperator {
        if (clientOrderIds.length != subaccountIds.length) revert
    LengthNotMatch();
        if (clientOrderIds.length != recipients.length) revert
    LengthNotMatch();
        if (clientOrderIds.length != expireTimes.length) revert
    LengthNotMatch();
        if (clientOrderIds.length != amounts.length) revert
    LengthNotMatch();
        if (clientOrderIds.length != fees.length) revert LengthNotMatch();
        if (clientOrderIds.length != signatures.length) revert
    LengthNotMatch();
        if (clientOrderIds.length != dstChainIds.length) revert
    LengthNotMatch();
```

```
        for (uint64 i = 0; i < subaccountIds.length; i++) {
            bytes32 user = subaccounts[subaccountIds[i]].chainAddress;
-           _userWithdraw(
-               clientOrderIds[i],
-               user,
-               recipients[i],
-               expireTimes[i],
-               dstChainIds[i],
-               amounts[i],
-               fees[i],
-               signatures[i],
-               false,
-               signatureType
-           );
+           try this.doUserWithdraw(
+               clientOrderIds[i],
+               user,
+               recipients[i],
+               expireTimes[i],
+               dstChainIds[i],
+               amounts[i],
+               fees[i],
+               signatures[i],
+               false,
+               signatureType
+           ) {} catch {}
        }
    }
+   function doUserWithdraw(
+        uint256 clientOrderId,
+        bytes32 user,
+        bytes32 recipient,
+        uint256 expireTime,
+        uint64 dstChainId,
+        uint256 amount,
+        uint256 fee,
+        bytes memory signatures,
+        bool isForce,
+        SignatureType signatureType
+    ) external {
+       require(msg.sender == address(this), "Asset: only self");
+        _userWithdraw(clientOrderId, user, recipient, expireTime,
   dstChainId, amount, fee, signatures, isForce, signatureType);
```

```
+     }
```

3.3 Lack of Access Control and Parameter Validation Enables Bypass ChainId

SEVERITY: HIGH **STATUS:** Fixed

PATH:

src/stargate/StargateWithdraw.sol

DESCRIPTION:

crossChainWithdraw is external without any access restrictions, allowing anyone to call it directly but not through Asset.sol. In crossChainWithdraw no validation of sendParam and dstChainId and dstAddress is match or not.

```
// src/stargate/StargateWithdraw.sol
    function crossChainWithdraw(
        uint256 clientOrderId,
        bytes32 user,
        uint256 amount,
        uint256 dstChainId,
        bytes32 dstAddress,
        address refundAddress,
        SendParam memory sendParam,
        MessagingFee memory messagingFee
    ) external payable nonReentrant validChain(dstChainId) returns
   (bytes32 guid) {
        // Get destination endpoint ID
        uint32 dstEid = chainIdToEndpointId[dstChainId];
        if (dstEid == 0) revert InvalidEndpointId();

        // Transfer USDC from caller to this contract
        USDC.safeTransferFrom(msg.sender, address(this), amount);

        // Approve Stargate pool to spend USDC
        USDC.forceApprove(address(stargate), amount);

        // Execute cross-chain send via Stargate with error handling
        try stargate.sendToken{
            value: msg.value
        }(
```

```
            sendParam, messagingFee, refundAddress
        ) returns (MessagingReceipt memory msgReceipt, OFTReceipt memory,
    Ticket memory) {
            // Success: Reset approval and emit success event
            USDC.forceApprove(address(stargate), 0);

            emit CrossChainWithdrawInitiated(
                clientOrderId, user, amount, block.chainid, dstEid,
    dstAddress, msgReceipt.guid
            );
            return msgReceipt.guid;
        // ...
     }
Any external caller can invoke crossChainWithdraw with malicious
    parameters.
// Attacker directly calls crossChainWithdraw
stargateWithdraw.crossChainWithdraw(
    clientOrderId: 123,
    user: victimUser,
    amount: 1000e6,
    dstChainId: 42161,                    // Arbitrum (passes validChain
    check)
    dstAddress: bytes32(legitAddress),    // Legitimate address
    refundAddress: attackerAddress,
    sendParam: SendParam({
        dstEid: 10,                       // Optimism endpoint (different
    chain)
        to: bytes32(attackerAddress),     // Attacker's address
        amountLD: 1000e6,
        ...
    }),
    messagingFee: maliciousFee
);

Result: Event shows "Arbitrum -> legitAddress", funds actually go to "
    Optimism -> attackerAddress"
```

IMPACT:

1. Event Spoofing & Whitelist Bypass: bypassing validChain can route funds to arbitrary chains/addresses while logs show a "safe" destination, deceiving off chain monitoring.

2. Total Logic Bypass: By bypassing Asset.sol and calling StargateWithdraw directly, attackers skip all critical high level validations (Signature verification, Expiry checks, clientOrderId deduplication, and User Balance checks), while still generating official events used for off chain accounting and settlement.

RECOMMENDATIONS:

To let off chain monitor right event and prevent bypass validChain, can add onlyAsset modifier.

```
// src/stargate/StargateWithdraw.sol
+    error OnlyAsset();
+    address public assetContract; // Set in constructor or via setter
+
+    modifier onlyAsset() {
+        if (msg.sender != assetContract) revert OnlyAsset();
+        _;
+    }
+
     function crossChainWithdraw(
         uint256 clientOrderId,
         bytes32 user,
         uint256 amount,
         uint256 dstChainId,
         bytes32 dstAddress,
         address refundAddress,
         SendParam memory sendParam,
         MessagingFee memory messagingFee
-    ) external payable nonReentrant validChain(dstChainId) returns
    (bytes32 guid) {
+    ) external payable nonReentrant onlyAsset validChain(dstChainId)
    returns (bytes32 guid) {
         // Get destination endpoint ID
         uint32 dstEid = chainIdToEndpointId[dstChainId];
```

3.4 Continuous batch updates can indefinitely delay force withdrawals

SEVERITY: MEDIUM **STATUS:** Acknowledge

PATH:

src/Asset.sol

DESCRIPTION:

The project design allows users to urgently retrieve tokens using forceWithdraw() after a 7-day time lock. This relies on lastBatchTime being updated in batchUpdate(). If SettlementOperator is called, it will cause a delay in retrieval. The problem is that batchUpdate() has no time limit for being called.

```
// src/Asset.sol
    function batchUpdate(
        uint256 batchId,
        int32 seqInBatch,
        uint256 antxChainHeight,
        BatchUpdateData memory batchUpdateData
    ) public onlySettlementOperator {
        // ...

        lastBatchId = batchId;
        batchSeqIds[batchId][seqInBatch] = true;
        lastBatchTime = block.timestamp;
        lastAntxChainHeight = antxChainHeight;
        emit BatchUpdated(batchId, antxChainHeight, block.timestamp);
    }
// src/Asset.sol
    function forceWithdraw(
        uint64 subaccountId,
        uint256 amount,
        uint256 expireTime,
        SignatureType signatureType,
        bytes memory signatures,
        uint64 dstChainId
    ) external nonReentrant validAmount(amount) {
        // check time lock
```

```
        if (block.timestamp < lastBatchTime + FORCE_WITHDRAW_TIME_LOCK)
    revert TimeLockNotPassed();
        // force withdraw
        bytes32 user = subaccounts[subaccountId].chainAddress;
        _userWithdraw(0, user, user, expireTime, dstChainId, amount, 0,
    signatures, true, signatureType);
        emit ForceWithdraw(user, user, amount, dstChainId);
     }
```

IMPACT:

When the interval between batchUpdate() calls is less than 7, the withdrawal will be delayed, resulting in the user's funds being unable to be withdrawn.

RECOMMENDATIONS:

Ensure that the execution interval of batchUpdate() is greater than the time lock.

3.5 Emergency withdraw signatures lack one-time use protection

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`src/Asset.sol`

DESCRIPTION:

The emergencyWithdraw() and emergencyWithdrawETH() functions employ a multi-signature mechanism to verify the authenticity of multiple signatures. However, _hashEmergencyWithdraw lacks a unique identifier when calculating the hash value. Furthermore, since emergencyWithdraw() lacks access control, this could lead to signature replay.

```
    function emergencyWithdraw(
        address token,
        address to,
        uint256 amount,
        uint256 expireTime,
        address[] memory allSigners,
        bytes[] memory signatures
    ) external nonReentrant validAddress(to) validAmount(amount) {
        // ...

        // verify multi signatures
        bytes32 operationHash = _hashEmergencyWithdraw(token, to, amount,
    expireTime);
        operationHash =
    MessageHashUtils.toEthSignedMessageHash(operationHash);

        for (uint8 index = 0; index < allSigners.length; index++) {
            address signer = ECDSA.recover(operationHash,
    signatures[index]);
            if (signer != allSigners[index]) revert InvalidSigner();
            if (!isAllowedSigner(signer)) revert NotAllowedSigner();
        }

        // ...
    }
```

```
    function _hashEmergencyWithdraw(address token, address to, uint256
amount, uint256 expireTime)
        internal
        view
        returns (bytes32)
    {
        return keccak256(
            abi.encodePacked("EMERGENCY_WITHDRAW", token, to, amount,
expireTime, address(this), block.chainid)
        );
    }
```

IMPACT:

An attacker can use an existing valid signature to transfer the contract's token to the address specified in the signature.

RECOMMENDATIONS:

Add a nonce parameter when signing to ensure uniqueness. Taking emergencyWithdraw() as an example, emergencyWithdrawETH() fixes the same issue.

```
+ uint256 public nonce;

    function emergencyWithdraw(
+       uint256 _nonce,
        address token,
        address to,
        uint256 amount,
        uint256 expireTime,
        address[] memory allSigners,
        bytes[] memory signatures
    ) external nonReentrant validAddress(to) validAmount(amount) {
        // ...
+       if (_nonce != nonce) revert InvalidNonce();

        // verify multi signatures
-       bytes32 operationHash = _hashEmergencyWithdraw(token, to, amount,
expireTime);
+       bytes32 operationHash = _hashEmergencyWithdraw(_nonce, token, to,
amount, expireTime);
```

```
        operationHash =
    MessageHashUtils.toEthSignedMessageHash(operationHash);

        for (uint8 index = 0; index < allSigners.length; index++) {
            address signer = ECDSA.recover(operationHash,
    signatures[index]);
            if (signer != allSigners[index]) revert InvalidSigner();
            if (!isAllowedSigner(signer)) revert NotAllowedSigner();
        }
+        nonce = nonce + 1;
        // ...
    }

-   function _hashEmergencyWithdraw(address token, address to, uint256
    amount, uint256 expireTime)
+   function _hashEmergencyWithdraw(uint256 nonce, address token, address
    to, uint256 amount, uint256 expireTime)
        internal
        view
        returns (bytes32)
    {
        return keccak256(
-           abi.encodePacked("EMERGENCY_WITHDRAW", token, to, amount,
    expireTime, address(this), block.chainid)
+           abi.encodePacked("EMERGENCY_WITHDRAW", nonce, token, to,
    amount, expireTime, address(this), block.chainid)
        );
    }
```

3.6 _userWithdraw did not charge a transaction fee

SEVERITY: MEDIUM **STATUS:** Acknowledge

PATH:

src/Asset.sol

DESCRIPTION:

The _userWithdraw function accepts a fee parameter as the transaction fee, but in the actual code, fee is only used as a parameter to generate a hash.

```
// src/Asset.sol
    function _userWithdraw(
        uint256 clientOrderId,
        bytes32 user,
        bytes32 recipient,
        uint256 expireTime,
        uint64 dstChainId,
        uint256 amount,
        uint256 fee,
        bytes memory signatures,
        bool isForce,
        SignatureType signatureType
    ) internal validAmount(amount) {
            // ...

            // check user signature
            bytes32 operationHash =
                _hashUserWithdraw(clientOrderId, user, recipient, amount,
   fee, expireTime, dstChainId);
            operationHash =
   MessageHashUtils.toEthSignedMessageHash(operationHash);

            // ...

    }
```

IMPACT:

Protocol may forgo intended withdrawal fees, causing recurring revenue loss and accounting mismatch.

RECOMMENDATIONS:

Design based on project requirements and decide whether to charge a fee.

3.7 Stargate Pool Token Mismatch Enables Fund Loss and Theft

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`src/stargate/StargateWithdraw.sol`

DESCRIPTION:

StargateWithdraw locks USDC as immutable in the constructor but allows the owner to change the stargate pool address at any time via setStargatePool. There is no validation that the new pool's underlying token matches USDC, creating a mismatch between the contract's token handling logic and the actual Stargate pool configuration.

```
// src/stargate/StargateWithdraw.sol
    IERC20 public immutable USDC;

    constructor(address _usdc, address _stargate, address _owner)
   Ownable(_owner) {
        if (_usdc == address(0)) revert InvalidChainId();
        if (_stargate == address(0)) revert InvalidStargatePool();

        USDC = IERC20(_usdc);
        stargate = IStargate(_stargate);
    }

    function setStargatePool(address _stargate) external onlyOwner {
        if (_stargate == address(0)) revert InvalidStargatePool();
        address oldPool = address(stargate);
        stargate = IStargate(_stargate);
        emit StargatePoolUpdated(oldPool, _stargate);
    }

    function crossChainWithdraw(
        // ...
    ) external payable nonReentrant validChain(dstChainId) returns
   (bytes32 guid) {
        // ...
        //@audit Always pulls USDC from caller
```

```
        USDC.safeTransferFrom(msg.sender, address(this), amount);

        //@audit Approves the current stargate pool which may not use USDC
        USDC.forceApprove(address(stargate), amount);

        //@audit Calls stargate.sendToken which may expect a different
    token
        try stargate.sendToken{value: msg.value}(sendParam, messagingFee,
    refundAddress) {
        // ...
     }
}
```

IMPACT:

If stargate is configured to a Native ETH pool, users will lose USDC stuck in contract while also paying ETH for the bridge. Led to double payment. The transaction appears successful, making the loss invisible until reconciliation.

RECOMMENDATIONS:

Add validation to ensure the Stargate pool's underlying token always matches the contract's USDC address.

```
// src/stargate/StargateWithdraw.sol
+     error InvalidPoolToken();

     constructor(address _usdc, address _stargate, address _owner)
    Ownable(_owner) {
        if (_usdc == address(0)) revert InvalidChainId();
        if (_stargate == address(0)) revert InvalidStargatePool();

        USDC = IERC20(_usdc);
        stargate = IStargate(_stargate);
+       if (stargate.token() != address(USDC)) revert InvalidPoolToken();
     }

      function setStargatePool(address _stargate) external onlyOwner {
          if (_stargate == address(0)) revert InvalidStargatePool();
+
+         IStargate newPool = IStargate(_stargate);
```

```
+         if (newPool.token() != address(USDC)) revert InvalidPoolToken();
+
          address oldPool = address(stargate);
-         stargate = IStargate(_stargate);
+         stargate = newPool;
          emit StargatePoolUpdated(oldPool, _stargate);
      }
```

3.8 riskTiers Unverified Ranking

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`src/margin/MarginAsset.sol`

DESCRIPTION:

The findPositionRiskTier() function searches for the first matching risk level in ascending order, but currently there is no explicit validation of whether riskTiers are sorted in ascending order, so the final result may not be as expected.

```
    /**
     * @notice Find corresponding risk tier based on position value
     * @param riskTiers Risk tier list (must be sorted by
positionValueUpperBound from small to large)
     * @param positionValueAbs Position absolute value
     * @return riskTierIndex Found risk tier index, if not found return
the last tier
     */
    function findPositionRiskTier(RiskTier[] memory riskTiers, uint256
positionValueAbs)
        internal
        pure
        returns (uint256 riskTierIndex)
    {
        require(riskTiers.length > 0, "risk tiers is empty");

        // Iterate to find matching risk tier
        for (uint256 i = 0; i < riskTiers.length; i++) {
            uint256 upperBound =
uint256(riskTiers[i].positionValueUpperBound);
            if (positionValueAbs <= upperBound) {
                return i;
            }
        }

        // Not found, return the last tier (fallback)
```

```
        return riskTiers.length - 1;
    }
```

IMPACT:

Incorrect ordering of riskTiers can cause the returned risk rating to be different from the expected one.

RECOMMENDATIONS:

The correct arrangement order was confirmed by iterating through riskTiers.positionValueUpperBound.

```
    function findPositionRiskTier(RiskTier[] memory riskTiers, uint256
   positionValueAbs)
        internal
        pure
        returns (uint256 riskTierIndex)
    {
        require(riskTiers.length > 0, "risk tiers is empty");

+       // must be sorted by positionValueUpperBound from small to large
+       for (uint256 i = 1; i < riskTiers.length; i++) {
+           require(
+               riskTiers[i].positionValueUpperBound >= riskTiers[i -
   1].positionValueUpperBound,
+               "risk tiers not sorted"
+           );
+       }

        // Iterate to find matching risk tier
        for (uint256 i = 0; i < riskTiers.length; i++) {
            uint256 upperBound =
   uint256(riskTiers[i].positionValueUpperBound);
            if (positionValueAbs <= upperBound) {
                return i;
            }
        }

        // Not found, return the last tier (fallback)
        return riskTiers.length - 1;
    }
```

3.9 Zero-initialized lastBatchTime enables immediate force withdrawals

SEVERITY: LOW **STATUS:** Fixed

PATH:

src/Asset.sol

DESCRIPTION:

The forceWithdraw() function calculates the time for emergency retrieval based on the updated lastBatchTime. If lastBatchTime is not initialized after deployment, its default value will be 0, allowing the time lock restriction to be bypassed.

```
// src/Asset.sol
    function forceWithdraw(
        uint64 subaccountId,
        uint256 amount,
        uint256 expireTime,
        SignatureType signatureType,
        bytes memory signatures,
        uint64 dstChainId
    ) external nonReentrant validAmount(amount) {
        // check time lock
        if (block.timestamp < lastBatchTime + FORCE_WITHDRAW_TIME_LOCK)
   revert TimeLockNotPassed();
        // force withdraw
        bytes32 user = subaccounts[subaccountId].chainAddress;
        _userWithdraw(0, user, user, expireTime, dstChainId, amount, 0,
   signatures, true, signatureType);
        emit ForceWithdraw(user, user, amount, dstChainId);
    }
```

IMPACT:

Since there is no batchUpdate() call after the contract is deployed, the user has no withdrawable balance, so the impact is minimal.

RECOMMENDATIONS:

It is recommended to initialize lastBatchTime in the initialize() operation.

```
+   bool public hasBatchUpdate;

    function forceWithdraw(
        uint64 subaccountId,
        uint256 amount,
        uint256 expireTime,
        SignatureType signatureType,
        bytes memory signatures,
        uint64 dstChainId
    ) external nonReentrant validAmount(amount) {
+       if (!hasBatchUpdate) revert NotInitLastBatchTime();
        // check time lock
        if (block.timestamp < lastBatchTime + FORCE_WITHDRAW_TIME_LOCK)
   revert TimeLockNotPassed();
        // force withdraw
        bytes32 user = subaccounts[subaccountId].chainAddress;
        _userWithdraw(0, user, user, expireTime, dstChainId, amount, 0,
   signatures, true, signatureType);
        emit ForceWithdraw(user, user, amount, dstChainId);
    }


    function batchUpdate(
        uint256 batchId,
        int32 seqInBatch,
        uint256 antxChainHeight,
        BatchUpdateData memory batchUpdateData
    ) public onlySettlementOperator {
        // ...

+       if (!hasBatchUpdate) {
+           hasBatchUpdate = true;
+       }

        lastBatchId = batchId;
        batchSeqIds[batchId][seqInBatch] = true;
        lastBatchTime = block.timestamp;
        lastAntxChainHeight = antxChainHeight;
        emit BatchUpdated(batchId, antxChainHeight, block.timestamp);
```

```
    }
```

3.10 Sign flip caused by uint256-to-int256 conversion overflow

SEVERITY: LOW **STATUS:** Fixed

PATH:

`src/margin/MarginAsset.sol`

DESCRIPTION:

The MarginAsset contract primarily handles asset calculations. However, some of its conversion logic from uint256 to int256 lacks adequate safety measures. For example, in getCrossTransferOutAvailableAmount, calculatePositionValue and calculateFundingAmount, using int256() to forcibly convert a uint256 number risks sign flipping if the value exceeds the maximum value of int256.

```
// src/margin/MarginAsset.sol
    function calculatePositionValue(
        int256 openSize,
        uint256 oraclePrice,
        uint32 stepSizeScale,
        uint32 tickSizeScale,
        uint32 coinStepSizeScale
    ) internal pure returns (int256 positionValue) {
        // Calculate (openSize * oraclePrice) / 10^(stepSizeScale +
   tickSizeScale - coinStepSizeScale)
        uint256 absOpenSize = absInt(openSize);
        uint256 value = absOpenSize * oraclePrice;
        // ...
        return openSize < 0 ? -int256(value) : int256(value);
    }
```

IMPACT:

When an extreme case occurs where a number exceeds the maximum value of int256, a forced cast to int256 will cause a sign flip, resulting in an unexpected calculation result.

RECOMMENDATIONS:

Add a maximum value check before the type cast to ensure it does not exceed the maximum value of int256.

```
+ require(value <= uint256(type(int256).max), "int256 conversion
    overflow");
```

3.11 Redundant batch length checks increase gas costs

SEVERITY: INFO **STATUS:** Fixed

PATH:

`src/Asset.sol`

DESCRIPTION:

The batchWithdraw() function has somewhat cumbersome logic when validating parameter lengths, and it also causes unnecessary gas waste.

```
    function batchWithdraw(
        // ...
    ) external nonReentrant onlyWithdrawOperator {
        if (clientOrderIds.length != subaccountIds.length) revert
    LengthNotMatch();
        if (clientOrderIds.length != recipients.length) revert
    LengthNotMatch();
        if (clientOrderIds.length != expireTimes.length) revert
    LengthNotMatch();
        if (clientOrderIds.length != amounts.length) revert
    LengthNotMatch();
        if (clientOrderIds.length != fees.length) revert LengthNotMatch();
        if (clientOrderIds.length != signatures.length) revert
    LengthNotMatch();
        if (clientOrderIds.length != dstChainIds.length) revert
    LengthNotMatch();

        // ...
    }
```

IMPACT:

Extra length checks waste gas in a frequently used batching path.

RECOMMENDATIONS:

Optimize the code to reduce the number of length calculations.

```
    function batchWithdraw(
        // ...
    ) external nonReentrant onlyWithdrawOperator {
-       if (clientOrderIds.length != subaccountIds.length) revert
    LengthNotMatch();
-       if (clientOrderIds.length != recipients.length) revert
    LengthNotMatch();
-       if (clientOrderIds.length != expireTimes.length) revert
    LengthNotMatch();
-       if (clientOrderIds.length != amounts.length) revert
    LengthNotMatch();
-       if (clientOrderIds.length != fees.length) revert LengthNotMatch();
-       if (clientOrderIds.length != signatures.length) revert
    LengthNotMatch();
-       if (clientOrderIds.length != dstChainIds.length) revert
    LengthNotMatch();

+       uint256 len = clientOrderIds.length;
+       if (
+           len != subaccountIds.length ||
+           len != amounts.length ||
+           len != fees.length ||
+           len != signatures.length ||
+           len != dstChainIds.length
+       ) revert LengthNotMatch();


        // ...
    }
```

3.12 Incorrect Error Type Used in Constructor

SEVERITY: INFO **STATUS:** Fixed

PATH:

`src/stargate/StargateWithdraw.sol`

DESCRIPTION:

When validating that _usdc is not a zero address, the code incorrectly uses InvalidChainId() error is semantically incorrect.

```
// src/stargate/StargateWithdraw.sol
    function _validChain(uint256 chainId) internal view {
        if (chainId == 0) revert InvalidChainId();
        if (chainId == block.chainid) {
            revert CrossChainNotSupported(chainId);
        }
        if (!supportedChains[chainId]) {
            revert CrossChainNotSupported(chainId);
        }
    }

    constructor(address _usdc, address _stargate, address _owner)
   Ownable(_owner) {
        if (_usdc == address(0)) revert InvalidChainId();
        if (_stargate == address(0)) revert InvalidStargatePool();

        USDC = IERC20(_usdc);
        stargate = IStargate(_stargate);
    }
```

IMPACT:

Using the wrong error type reduces clarity for integrators and offchain monitoring.

RECOMMENDATIONS:

Use a more appropriate error type.

```
// src/stargate/StargateWithdraw.sol
    error InvalidChainId();
    error CrossChainNotSupported(uint256 chainId);
    error InvalidStargatePool();
+   error InvalidUSDCAddress();
    error InsufficientBalance();
    error TransferFailed();
    error InvalidEndpointId();
    error RefundFailed();

    constructor(address _usdc, address _stargate, address _owner)
    Ownable(_owner) {
-       if (_usdc == address(0)) revert InvalidChainId();
+       if (_usdc == address(0)) revert InvalidUSDCAddress();
        if (_stargate == address(0)) revert InvalidStargatePool();

        USDC = IERC20(_usdc);
        stargate = IStargate(_stargate);
    }
```

3.13 Antex naming error

SEVERITY: INFO **STATUS:** Acknowledge

PATH:

`src/Asset.sol`

DESCRIPTION:

The Asset.sol contract has many naming errors, Antx should be Antex.

RECOMMENDATIONS:

Rename Antx to Antex across the codebase and update any related documentation and events.

4. CONCLUSION

In this audit, we thoroughly analyzed **Antx** smart contract implementation. The problems found are described and explained in detail in Section 3. The problems found in the audit have been communicated to the project leader. We therefore consider the audit result to be **PASSED**.

To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5. APPENDIX

5.1 Basic Coding Assessment

5.1.1 Apply Verification Control

Description	The security of apply verification
Result	Not found
Severity	CRITICAL

5.1.2 Authorization Access Control

Description	Permission checks for external integral functions
Result	Not found
Severity	CRITICAL

5.1.3 Forged Transfer Vulnerability

Description	Assess whether there is a forged transfer notification vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.4 Transaction Rollback Attack

Description	Assess whether there is transaction rollback attack vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.5 Transaction Block Stuffing Attack

Description	Assess whether there is transaction blocking attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.6 Soft Fail Attack Assessment

Description	Assess whether there is soft fail attack vulnerability
Result	Not found
Severity	CRITICAL

6. DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without ExVul's prior written consent.

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 ExVul 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. ExVul's position is that each company and individual are responsible for their own due diligence and continuous security. ExVul's 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.

7. REFERENCES

[1] MITRE. CWE-191: Integer Underflow (Wrap or Wraparound). https://cwe.mitre.org/data/definitions/191.html.

[2] MITRE. CWE-197: Numeric Truncation Error. https://cwe.mitre.org/data/definitions/197.html.

[3] MITRE. CWE-400: Uncontrolled Resource Consumption. https://cwe.mitre.org/data/definitions/400.html.

[4] MITRE. CWE-440: Expected Behavior Violation. https://cwe.mitre.org/data/definitions/440.html.

[5] MITRE. CWE-684: Protection Mechanism Failure. https://cwe.mitre.org/data/definitions/693.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE CATEGORY: Resource Management Errors. https://cve.mitre.org/data/definitions/399.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

8. About Exvul Security

Premier Security for the Web3 Ecosystem

ExVul is a premier Web3 security firm committed to forging a secure and trustworthy decentralized ecosystem. Our elite team consists of security veterans from world-leading technology and blockchain security firms, including Huawei, YBB Captical, Qihoo 360, Amber, ByteDance, MoveBit, and PeckShield. Team member Nolan is ranked as a top-40 whitehat on Immunefi and is the platform's sole All-Star in the APAC region.

Our expertise covers the full spectrum of Web3 security. We conduct **meticulous smart contract audits**, having fortified thousands of projects on chains like Evm, Solana, Aptos, Sui etc. Our **Blockchain Protocol Audits** secure the core infrastructure of L1/L2 by uncovering deep-seated vulnerabilities. We also offer **comprehensive wallet audits** to protect user assets and provide **proactive web3 pentest**, enabling partners to neutralize threats before they strike.

Trusted by industry leaders, ExVul is the security partner for **OKX, Bitget, Cobo, Infini, Stacks, Aptos, Sui, CoreDAO, Sei** etc.

Contact

Website
www.exvul.com

Email
contact@exvul.com

Twitter
@EXVULSEC

Github
github.com/EXVUL-Sec

ExVul