# Competitive Security Assessment

## NeoX_Bridge_Contract

Aug 8th, 2024

Secure3

secure3.io

# Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.
The comprehensive examination and auditing scope includes:
• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
• Verify the code base is compliant with the most up-to-date industry standards and security best practices.
• Comprehensive line-by-line manual code review of the entire codebase by industry experts.
The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

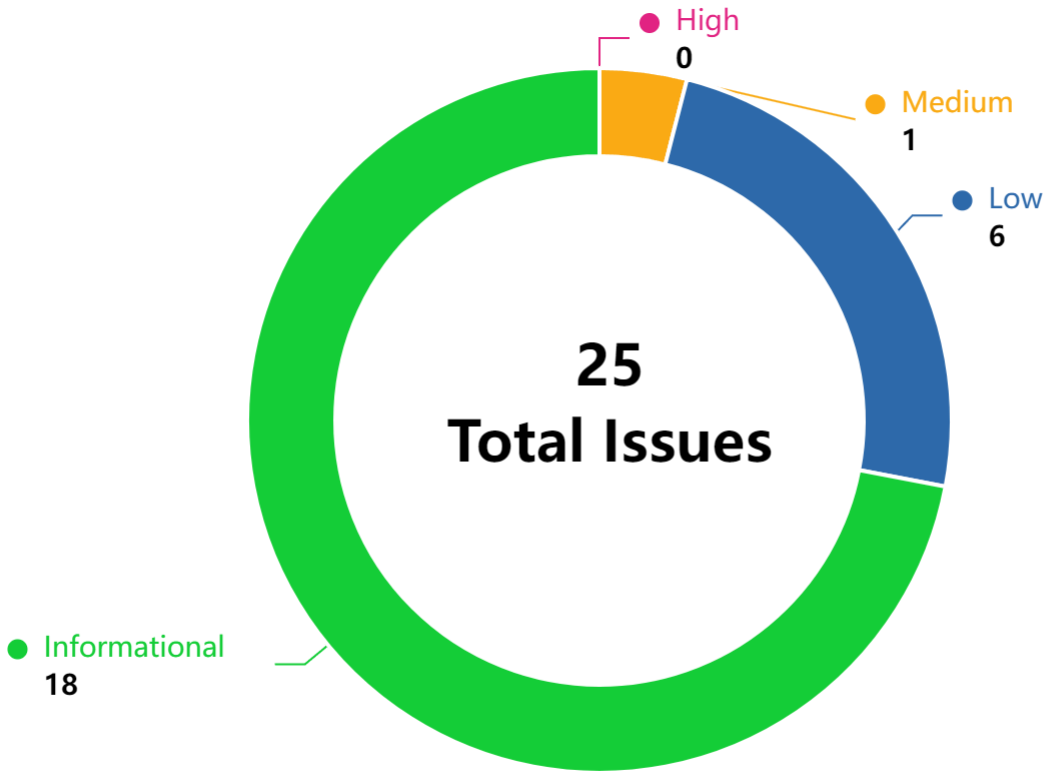
# Overview

| | |
|---|---|
| Project Name | NeoX_Bridge_Contract |
| Language | solidity |
| Codebase | • https://github.com/bane-labs/bridge-evm-contracts/tree/develop<br>• audit version - f4a3a39d62cf16e403bf669172d55f3d13e4083c<br>• final version - 06147506c713f78cfd6bd9e914709ba2c22a4608 |

# Audit Scope

| File | SHA256 Hash |
| --- | --- |
| contracts/bridge/BridgeImpl.sol | 8240722f4842acc50bc5eb2cd8507903bc4ec91c00c6ef354f89e36f9d93ae8e |
| contracts/bridge/BridgeStorage.sol | 2c79bcd4d1950bd48c31dcb025ec1382e8b35b23132b88caa4bba072633e1ada |
| contracts/management/BridgeManagementImpl.sol | 45c76319a6f65933d1faadb689c184fe4cd3d8e2be7b088eff93f36b4ad9d5e4 |
| contracts/management/BridgeManagementStorage.sol | de6e984aa9ccdc1f0823331352e3b8731728fd2eb52a1a6b73995a8a9ba8631e |
| contracts/library/TokenBridgeLib.sol | 4059c0a26781c34bdba641ffde1bd95aa0adb8d7980149f1f67f420ffef12980 |
| contracts/library/StorageTypes.sol | 932e91800fff56351026701f78ca8192507dc75a4348470e5b98743fd5584ac0 |
| contracts/library/GasBridgeLib.sol | 48b931e2f73ca01b38fd2d7c4e9a416746211f376ee72737e465fa46ff4653de |
| contracts/library/BridgeLib.sol | a1cc4a08dd5c0494700eb85366e8dc7fbecd629386682ad029b0b0addcbd7b16 |
| contracts/bridge/BridgeStorageV1.sol | 0f55145c232abc3d9e01e237a532c42f5c41d838bcf6c0c531a553688f5fe3e0 |
| contracts/library/ManagementLib.sol | 2a51a9ea7ae0e9a6d60cde18d82ce47c57311f1a67a8f91440e7f51f6dfbb26c |
| contracts/management/BridgeManagementStorageV1.sol | 912e6c5144acd863af1cb78de76e12e543d666f3579dc865ec786e4f84594ea4 |

# Code Assessment Findings




| ID | Name | Category | Severity | Client Response | Contributor |
|---|---|---|---|---|---|
| NEO-1 | Token Registration and Deregistration Confusion | Logical | Medium | Fixed | *** |
| NEO-2 | use of safeTransferFrom instead of transferFrom | Language Specific | Low | *** | |
| NEO-3 | Use of ecrecover restricts validators to be EOAs | Logical | Low | Acknowledged | *** |
| NEO-4 | Use of ecrecover is Susceptible to Signature Malleability | Signature Forgery or Replay | Low | Fixed | *** |
| NEO-5 | UUPS proxy contracts lack initialize function and be initialized in constructor | DOS | Low | Fixed | *** |
| NEO-6 | The value of `minAmount` should not be equal to the value of `maxAmount`. | Logical | Low | Fixed | *** |
| NEO-7 | Inconsistent balance on transfer-on-fee or deflationary tokens | Logical | Low | Fixed | *** |

| | | | | | |
|---|---|---|---|---|---|
| NEO-8 | msg.value exceeding the required fee is lost by the user | Logical | Informational | Fixed | *** |
| NEO-9 | Validator threshold can be set to 1 and malicious validator can forge Merkle root | Privilege Related | Informational | Fixed | *** |
| NEO-10 | Unused error: `TokenWithdrawalFailed` | Logical | Informational | Fixed | *** |
| NEO-11 | Unlocked Pragma Version | Code Style | Informational | Fixed | *** |
| NEO-12 | Threshold Condition in verifyValidatorSignatures Function | Privilege Related | Informational | Acknowledged | *** |
| NEO-13 | Support add, remove and replace functionality for validators | Code Style | Informational | Acknowledged | *** |
| NEO-14 | Solidity version 0.8.20+ may not work on other chains due to `PUSH0` | Language Specific | Informational | Acknowledged | *** |
| NEO-15 | Signature replay attack is possible for BridgeManagementImpl.verifyValidatorSignatures() | Signature Forgery or Replay | Informational | Acknowledged | *** |
| NEO-16 | Prevent underflow and provide clear feedback for insufficient fee | Privilege Related | Informational | Fixed | *** |
| NEO-17 | Ownership change should use two-step process | Privilege Related | Informational | Fixed | *** |
| NEO-18 | No not `0` check | Logical | Informational | Fixed | *** |
| NEO-19 | Missing data length check in `_executeERC20Transfer`, potential silent failures can occur | Logical | Informational | *** | |
| NEO-20 | Missing Zero Address Check | Logical | Informational | Fixed | *** |
| NEO-21 | Lack of check whether the token is registered | Logical | Informational | Fixed | *** |

| | | | | | |
|---|---|---|---|---|---|
| NEO-23 | Better way to do the double for loop in BridgeManagementImpl.verifyValidatorSignatures() | Code Style | Informational | Acknowledged | *** |
| NEO-24 | Attacker can abuse deposit nonce due to uint8 downcasting in BridgeLib._subsequentNonces(), users can lose fund | Language Specific | Informational | Fixed | *** |
| NEO-25 | Accumulated variable `unclaimedRewards` that are not actually used | Logical | Informational | Acknowledged | *** |

# NEO-1:Token Registration and Deregistration Confusion

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Medium | Fixed | *** |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L228-L250

```
228: function registerToken(
229:         address _neoXToken,
230:         StorageTypes.TokenConfig calldata _tokenConfig
231:     ) external override onlyGovernor {
232:         if (_neoXToken == address(0)) revert InvalidTokenAddress();
233:         if (_tokenConfig.minAmount > _tokenConfig.maxAmount)
234:             revert InvalidAmount();
235:         if (_tokenConfig.neoN3Token == address(0)) revert InvalidAddress();
236:
237:         _registerToken(_neoXToken, _tokenConfig);
238:         emit TokenRegister(_neoXToken, _tokenConfig);
239:     }
240:
241:     /**
242:      * @notice Unregister a token bridge.
243:      * @param _neoXToken the address of the token on the Neo X network.
244:      */
245:     function unregisterToken(
246:         address _neoXToken
247:     ) external override onlyGovernor onlyTokenBridgePaused(_neoXToken) {
```

```
248:         _unregisterToken(_neoXToken);
249:         emit TokenUnregister(_neoXToken, _getNeoN3Token(_neoXToken));
250:     }
```

- code/contracts/bridge/BridgeStorage.sol#L211-L242

```
211: function _registerToken(
212:         address _neoXToken,
213:         StorageTypes.TokenConfig memory _tokenConfig
214:     ) internal {
215:         // Check if token bridge is already registered
216:         if (_isRegisteredToken(_neoXToken))
217:             revert TokenBridgeAlreadyRegistered(_neoXToken);
218:         if (!TokenBridgeLib._isValidConfig(_tokenConfig))
219:             revert InvalidTokenConfig();
220:
221:         // Add token bridge to storage
222:         tokenBridges[_neoXToken] = StorageTypes.TokenBridge({
223:             paused: false,
224:             depositState: StorageTypes.State({nonce: 0, root: 0x0}),
225:             withdrawalState: StorageTypes.State({nonce: 0, root: 0x0}),
226:             config: _tokenConfig
227:         });
228:     }
229:
230:     function _isRegisteredToken(
```

```
231:         address _neoXToken
232:     ) internal view returns (bool) {
233:         return tokenBridges[_neoXToken].config.neoN3Token != address(0);
234:     }
235:
236:     function _unregisterToken(address _neoXToken) internal {
237:         if (!_isRegisteredToken(_neoXToken))
238:             revert TokenBridgeNotRegistered(_neoXToken);
239:         delete tokenBridges[_neoXToken];
240:         // If a token is unregistered, the claimables remain in storage.
241:         // This means, that they are locked, and can only ever be retrieved again if there's a new token
bridge registration with the same Neo X token address.
242:     }
```

## Description

***: Let's take two examples below

### `claimToken` Call after Deregistration

Calling process `registerToken->depositToken->pauseTokenBridge->unregisterToken->claimToken`
When calling the `unregisterToken` function to cancel a token, although the corresponding token entry in `tokenBridges` is deleted, the record in `tokenClaimables` still exists. This means that users can still call the `claimToken` function to claim the deposited but unallocated token. Because the `tokenBridges` data is a `mapping` type, the `onlyTokenBridgeUnpaused` check can pass

### `depositToken` Call after Token Reregistration

If a token is re-registered after being cancelled, there may be problems when executing the depositToken function. After the unregisterToken function is called, the entry in tokenBridges is deleted, and a new entry is created when re-registering, and the state is reinitialized. Since the `depositToken` function relies on the state information of the token (such as `depositState` and config), after re-registration, these state information are reset, which may cause inconsistent or incorrect verification of deposit data. For example, the re-registered token may not match the previous state, resulting in deposit verification failure or tampering of the deposit state.
Steps to Reproduce
Call `registerToken` to register a new token.
Call `depositToken` to deposit a token.
Call `pauseTokenBridge` to pause the token bridge.
Call `unregisterToken` to unregister the token.
Call `registerToken` again to re-register the same token.
Call `depositToken` again to deposit a token.
***: The `_unregisterToken` function will delete all data related to this token:

```
function _unregisterToken(address _neoXToken) internal {
        if (!_isRegisteredToken(_neoXToken))
            revert TokenBridgeNotRegistered(_neoXToken);
        delete tokenBridges[_neoXToken];
        // If a token is unregistered, the claimables remain in storage.
        // This means, that they are locked, and can only ever be retrieved again if there's a new token bri
dge registration with the same Neo X token address.
    }
```

If this unregistered token is registered again in the future, the root will be reset to `0x0` and the nonce will be reset to 0:

```
function _registerToken(
        address _neoXToken,
        StorageTypes.TokenConfig memory _tokenConfig
    ) internal {
        // Check if token bridge is already registered
        if (_isRegisteredToken(_neoXToken))
            revert TokenBridgeAlreadyRegistered(_neoXToken);
        if (!TokenBridgeLib._isValidConfig(_tokenConfig))
            revert InvalidTokenConfig();

        // Add token bridge to storage
        tokenBridges[_neoXToken] = StorageTypes.TokenBridge({
            paused: false,
            depositState: StorageTypes.State({nonce: 0, root: 0x0}),
            withdrawalState: StorageTypes.State({nonce: 0, root: 0x0}),
            config: _tokenConfig
        });
    }
```

This will lead to replay attacks. All deposits made before the token is unregistered can be executed again by calling `depositToken`.

## Recommendation

***: Improve the deregistration logic. When deregistering a token, all related `tokenClaimables` records should be cleaned up to prevent the claimToken from being called after deregistration. When re-registering a token, ensure that its status is consistent with the previous deregistration, or clearly distinguish the status of the new and old tokens to avoid deposit verification failures caused by status resets.
***: Consider using a flag to recognize whether this token is registered.

## Client Response

client response : Fixed. This issue has been fixed in commit a145664be1237c620896d65a85275926c701a2b6 by disallowing de-registration of token bridges.
client response : Fixed. This issue has been fixed in commit a145664be1237c620896d65a85275926c701a2b6 by disallowing de-registration of token bridges in the first place.

# NEO-2:use of safeTransferFrom instead of transferFrom

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Language Specific | Low | *** | |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L442-L505
- code/contracts/bridge/BridgeImpl.sol#L443-L506
- code/contracts/bridge/BridgeImpl.sol#L466
- code/contracts/bridge/BridgeImpl.sol#L466-L470
- code/contracts/bridge/BridgeImpl.sol#L466-470

```
442: */
443:     function withdrawToken(
444:         address _neoXToken,
445:         address _to,
446:         uint256 _amount
447:     )
448:         external
449:         payable
450:         override
451:         nonReentrant
452:         onlyBridgeUnpaused
453:         onlyTokenBridgeUnpaused(_neoXToken)
454:     {
455:         if (!_isRegisteredToken(_neoXToken))
456:             revert TokenBridgeNotRegistered(_neoXToken);
457:         StorageTypes.TokenConfig memory config = _getTokenConfig(_neoXToken);
458:         uint256 tokenValue = _amount;
459:         if (tokenValue < config.minAmount) revert InvalidAmount();
460:         if (tokenValue > config.maxAmount) revert InvalidAmount();
461:         if (msg.value < config.fee)
```

```
462:             revert InsufficientFee(msg.value, config.fee);
463:         _addUnclaimedRewards(msg.value);
464:
465:         // Execute the transfer of the tokens from the sender to the bridge contract.
466:         bool success = IERC20(_neoXToken).transferFrom(
467:             msg.sender,
468:             address(this),
469:             _amount
470:         );
471:         if (!success) revert TransferFailed();
472:
473:         // Compute the new root and update the token withdrawal state.
474:         StorageTypes.State memory state = _getTokenWithdrawalState(_neoXToken);
475:         uint256 newNonce = state.nonce + 1;
476:
477:         if (config.executionType == StorageTypes.ExecutionType.NEO) {
478:             if (tokenValue % 1e18 != 0) {
479:                 revert InvalidAmount();
480:             }
481:             tokenValue /= 1e18;
```

```
482:         }
483:
484:         bytes32 withdrawalHash = TokenBridgeLib._hashTokenBridgeOp(
485:             config.neoN3Token,
486:             _neoXToken,
487:             newNonce,
488:             _to,
489:             tokenValue
490:         );
491:         bytes32 newRoot = BridgeLib._computeNewRoot(state.root, withdrawalHash);
492:         _setTokenWithdrawalState(
493:             _neoXToken,
494:             StorageTypes.State({nonce: newNonce, root: newRoot})
495:         );
496:         emit TokenWithdrawal(
497:             _neoXToken,
498:             config.neoN3Token,
499:             newNonce,
500:             _to,
501:             tokenValue,
```

```
502:             msg.sender,
503:             withdrawalHash,
504:             newRoot
505:         );
```

```
443: function withdrawToken(
444:         address _neoXToken,
445:         address _to,
446:         uint256 _amount
447:     )
448:         external
449:         payable
450:         override
451:         nonReentrant
452:         onlyBridgeUnpaused
453:         onlyTokenBridgeUnpaused(_neoXToken)
454:     {
455:         if (!_isRegisteredToken(_neoXToken))
456:             revert TokenBridgeNotRegistered(_neoXToken);
457:         StorageTypes.TokenConfig memory config = _getTokenConfig(_neoXToken);
458:         uint256 tokenValue = _amount;
459:         if (tokenValue < config.minAmount) revert InvalidAmount();
460:         if (tokenValue > config.maxAmount) revert InvalidAmount();
461:         if (msg.value < config.fee)
462:             revert InsufficientFee(msg.value, config.fee);
```

```
463:         _addUnclaimedRewards(msg.value);
464:
465:         // Execute the transfer of the tokens from the sender to the bridge contract.
466:         bool success = IERC20(_neoXToken).transferFrom(
467:             msg.sender,
468:             address(this),
469:             _amount
470:         );
471:         if (!success) revert TransferFailed();
472:
473:         // Compute the new root and update the token withdrawal state.
474:         StorageTypes.State memory state = _getTokenWithdrawalState(_neoXToken);
475:         uint256 newNonce = state.nonce + 1;
476:
477:         if (config.executionType == StorageTypes.ExecutionType.NEO) {
478:             if (tokenValue % 1e18 != 0) {
479:                 revert InvalidAmount();
480:             }
481:             tokenValue /= 1e18;
482:         }
```

```
483:
484:         bytes32 withdrawalHash = TokenBridgeLib._hashTokenBridgeOp(
485:             config.neoN3Token,
486:             _neoXToken,
487:             newNonce,
488:             _to,
489:             tokenValue
490:         );
491:         bytes32 newRoot = BridgeLib._computeNewRoot(state.root, withdrawalHash);
492:         _setTokenWithdrawalState(
493:             _neoXToken,
494:             StorageTypes.State({nonce: newNonce, root: newRoot})
495:         );
496:         emit TokenWithdrawal(
497:             _neoXToken,
498:             config.neoN3Token,
499:             newNonce,
500:             _to,
501:             tokenValue,
502:             msg.sender,
```

```
503:             withdrawalHash,
504:             newRoot
505:         );
506:     }
```

```
466: bool success = IERC20(_neoXToken).transferFrom(
```

```
466: bool success = IERC20(_neoXToken).transferFrom(
467:             msg.sender,
468:             address(this),
469:             _amount
470:         );
```

```
466: bool success = IERC20(_neoXToken).transferFrom(
467:             msg.sender,
468:             address(this),
469:             _amount
470:         );
```

## Description

***:

## Vulnerability Detail

The ERC20.transfer() and ERC20.transferFrom() functions return a boolean value indicating success. This parameter needs to be checked for success. Some tokens do not revert if the transfer failed but return false instead.

```
function withdrawToken(
    address _neoXToken,
    address _to,
    uint256 _amount
)
    external
    payable
    override
    nonReentrant
    onlyBridgeUnpaused
    onlyTokenBridgeUnpaused(_neoXToken)
{
    if (!_isRegisteredToken(_neoXToken))
        revert TokenBridgeNotRegistered(_neoXToken);
    StorageTypes.TokenConfig memory config = _getTokenConfig(_neoXToken);
    uint256 tokenValue = _amount;
    if (tokenValue < config.minAmount) revert InvalidAmount();
    if (tokenValue > config.maxAmount) revert InvalidAmount();
    if (msg.value < config.fee)
        revert InsufficientFee(msg.value, config.fee);
    _addUnclaimedRewards(msg.value);

    // Execute the transfer of the tokens from the sender to the bridge contract.
    bool success = IERC20(_neoXToken).transferFrom(
        msg.sender,
        address(this),
        _amount
    );
    if (!success) revert TransferFailed();

    // Compute the new root and update the token withdrawal state.
    StorageTypes.State memory state = _getTokenWithdrawalState(_neoXToken);
    uint256 newNonce = state.nonce + 1;

    if (config.executionType == StorageTypes.ExecutionType.NEO) {
        if (tokenValue % 1e18 != 0) {
            revert InvalidAmount();
        }
        tokenValue /= 1e18;
    }

    bytes32 withdrawalHash = TokenBridgeLib._hashTokenBridgeOp(
        config.neoN3Token,
        _neoXToken,
        newNonce,
        _to,
        tokenValue
```

Some tokens (like USDT) don't correctly implement the EIP20 standard and their transfer/ transferFrom function return void instead of a success boolean. Calling these functions with the correct EIP20 function signatures will always revert.

## Impact

Tokens that don't actually perform the transfer and return false are still counted as a correct transfer and tokens that don't correctly implement the latest EIP20 spec, like USDT, will be unusable in the protocol as they revert the transaction because of the missing return value.
***: `transferFrom` Method: This is the standard ERC20 method for transferring tokens from one account to another, assuming the sender has sufficient allowance. However, this method returns a boolean indicating success or failure, which is being checked in the code.
Potential Issues: Using `transferFrom` directly might not handle non-standard ERC20 tokens that do not return a boolean value correctly
***:
Take a look at https://github.com/Secure3Audit/code_NeoX_Bridge_Contract/blob/f3764f6770d85e9438997630adac73f5314afbf4/code/contracts/bridge/BridgeImpl.sol#L442-L505

```
function withdrawToken(
    address _neoXToken,
    address _to,
    uint256 _amount
)
    external
    payable
    override
    nonReentrant
    onlyBridgeUnpaused
    onlyTokenBridgeUnpaused(_neoXToken)
{
    if (!_isRegisteredToken(_neoXToken))
        revert TokenBridgeNotRegistered(_neoXToken);
    StorageTypes.TokenConfig memory config = _getTokenConfig(_neoXToken);
    uint256 tokenValue = _amount;
    if (tokenValue < config.minAmount) revert InvalidAmount();
    if (tokenValue > config.maxAmount) revert InvalidAmount();
    if (msg.value < config.fee)
        revert InsufficientFee(msg.value, config.fee);
    _addUnclaimedRewards(msg.value);

    // Execute the transfer of the tokens from the sender to the bridge contract.
    //@audit
    bool success = IERC20(_neoXToken).transferFrom(
        msg.sender,
        address(this),
        _amount
    );
    if (!success) revert TransferFailed();

    // Compute the new root and update the token withdrawal state.
    StorageTypes.State memory state = _getTokenWithdrawalState(_neoXToken);
    uint256 newNonce = state.nonce + 1;

    if (config.executionType == StorageTypes.ExecutionType.NEO) {
        if (tokenValue % 1e18 != 0) {
            revert InvalidAmount();
        }
        tokenValue /= 1e18;
    }

    bytes32 withdrawalHash = TokenBridgeLib._hashTokenBridgeOp(
        config.neoN3Token,
        _neoXToken,
        newNonce,
        _to,
```

This function is used to withdraw tokens to Neo N3 and it requires that the sender has approved the provided amount to the bridge contract, issue however is that when transferring the execution uses the classic `IERC20.transferFrom()`, however for tokens like USDT this attempt would never work, cause the token's transferFrom doesn't return a bool, see reasoning below:
Going to Openzeppelin's ERC20's `approve/transfer/transferFrom()` implementation we can see them specified like this https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol#L41-L78

```
    function transfer(address to, uint256 value) external returns (bool);
    (...snip)
    function approve(address spender, uint256 value) external returns (bool);
    (...snip)
    function transferFrom(address from, address to, uint256 value) external returns (bool);
```

But see `L199, L340 & L344` from `USDT`'s contract https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d831ec7#code
We can see that the definition of the functions are

```
    function approve(address _spender, uint _value) public
(...snip)
    function transfer(address _to, uint _value) public
(...snip)
    function transferFrom(address _from, address _to, uint _value)
```

This will then lead to different opcodes compiled by the compiler: when checking the length of the return data, `usdt.approve/transfer/transferFrom()` requires the length of the return data to be 0, while `ERC20.approve/transfer/transferFrom()` requires the length of the return data to be 1. Therefore, the **tx always reverts**.

***: As per ERC20 standard, the `transferFrom` should return a bool value. However, some tokens like `USDT` do not return a value. As a result the return data will always be false if you call the `transferFrom` function.
In `withdrawToken` function, it will call `IERC20(_neoXToken).transferFrom`:

```
// Execute the transfer of the tokens from the sender to the bridge contract.
        bool success = IERC20(_neoXToken).transferFrom(
            msg.sender,
            address(this),
            _amount
        );
        if (!success) revert TransferFailed();
```

If the `_neoXToken` is a USDT-like token, the function will always revert. These `_neoXToken` can not be bridged back to Neo N3 network.
***: The return value of the `transferFrom()` call is not checked.

## Recommendation

***: Recommend using OpenZeppelin's SafeERC20 with the `safeTransfer` and `safeTransferFrom` functions that handle the return value check as well as non-standard-compliant tokens.

```
    function withdrawToken(
        address _neoXToken,
        address _to,
        uint256 _amount
    )
        external
        payable
        override
        nonReentrant
        onlyBridgeUnpaused
        onlyTokenBridgeUnpaused(_neoXToken)
    {
        if (!_isRegisteredToken(_neoXToken))
            revert TokenBridgeNotRegistered(_neoXToken);
        StorageTypes.TokenConfig memory config = _getTokenConfig(_neoXToken);
        uint256 tokenValue = _amount;
        if (tokenValue < config.minAmount) revert InvalidAmount();
        if (tokenValue > config.maxAmount) revert InvalidAmount();
        if (msg.value < config.fee)
            revert InsufficientFee(msg.value, config.fee);
        _addUnclaimedRewards(msg.value);

        // Execute the transfer of the tokens from the sender to the bridge contract.
-        bool success = IERC20(_neoXToken).transferFrom(
+        bool success = IERC20(_neoXToken).safeTransferFrom(
            msg.sender,
            address(this),
            _amount
        );
        if (!success) revert TransferFailed();

        // Compute the new root and update the token withdrawal state.
        StorageTypes.State memory state = _getTokenWithdrawalState(_neoXToken);
        uint256 newNonce = state.nonce + 1;

        if (config.executionType == StorageTypes.ExecutionType.NEO) {
            if (tokenValue % 1e18 != 0) {
                revert InvalidAmount();
            }
            tokenValue /= 1e18;
        }

        bytes32 withdrawalHash = TokenBridgeLib._hashTokenBridgeOp(
            config.neoN3Token,
            _neoXToken,
            newNonce,
            _to,
```

***:** Utilizing a safe transfer library such as OpenZeppelin's SafeERC20 can provide better security and handle edge cases with non-standard tokens.
***:** Consider inheriting and using openzeppelin's `SafeERC20` instead, since the safe wrappers call this function on a low level and as such it would not lead to a revert.
***:** Consider using openzeppelin's `SafeERC20` instead.
***:** Since some ERC-20 tokens return no values and others return a `bool` value, they should be handled with care. We advise using the OpenZeppelin's `SafeERC20.sol` implementation to interact with the `transfer()` and `transferFrom()` functions of external ERC-20 tokens. The OpenZeppelin implementation checks for the existence of a return value and reverts if `false` is returned, making it compatible with all ERC-20 token implementations.

## Client Response

client response :

# NEO-3:Use of ecrecover restricts validators to be EOAs

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Low | Acknowledged | *** |

## Code Reference

- code/contracts/management/BridgeManagementImpl.sol#L29-L64

```
29: function verifyValidatorSignatures(
30:         bytes32 _newDepositRoot,
31:         BridgeLib.Signature[] calldata _signatures
32:     ) external view returns (bool) {
33:         uint256 threshold = validatorThreshold;
34:         if (_signatures.length != threshold) {
35:             return false;
36:         }
37:         bytes32 signedRootMsg = keccak256(
38:             abi.encodePacked(
39:                 "\x19Ethereum Signed Message:\n32",
40:                 keccak256(abi.encodePacked(_newDepositRoot))
41:             )
42:         );
43:         address[] memory recovered = new address[](threshold);
44:         for (uint i = 0; i < threshold; i++) {
45:             BridgeLib.Signature calldata sig = _signatures[i];
46:             recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
47:         }
48:         // check if all recovered addresses are in the validator set
```

```
49:         uint covered = 0;
50:         uint n = 0;
51:         uint j;
52:         uint validatorsLength = validators.length;
53:         for (uint i = 0; i < threshold; i++) {
54:             for (j = n; j < validatorsLength; j++) {
55:                 if (recovered[i] == validators[j]) {
56:                     covered++;
57:                     break;
58:                 }
59:             }
60:             n = j + 1;
61:         }
62:         return covered == threshold;
63:     }
```

## Description

***:

Take a look at https://github.com/Secure3Audit/code_NeoX_Bridge_Contract/blob/f3764f6770d85e9438997630adac73f5314afbf4/code/contracts/management/BridgeManagementImpl.sol#L29-L64

```
function verifyValidatorSignatures(
    bytes32 _newDepositRoot,
    BridgeLib.Signature[] calldata _signatures
) external view returns (bool) {
    uint256 threshold = validatorThreshold;
    if (_signatures.length != threshold) {
        return false;
    }
    bytes32 signedRootMsg = keccak256(
        abi.encodePacked(
            "\x19Ethereum Signed Message:\n32",
            keccak256(abi.encodePacked(_newDepositRoot))
        )
    );
    address[] memory recovered = new address[](threshold);
    for (uint i = 0; i < threshold; i++) {
        BridgeLib.Signature calldata sig = _signatures[i];
        recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
    }
    // check if all recovered addresses are in the validator set
    uint covered = 0;
    uint n = 0;
    uint j;
    uint validatorsLength = validators.length;
    for (uint i = 0; i < threshold; i++) {
        for (j = n; j < validatorsLength; j++) {
            if (recovered[i] == validators[j]) {
                covered++;
                break;
            }
        }
        n = j + 1;
    }
    return covered == threshold;
}
```

This function is used to verify the validator signatures, it uses the classic `ecrecover`, issue however is that it doesn't take into account that validators could aswell be contracts in the case protocol plans to scale, if this happens, the signatures can never be verified from these contracts, considering using `ecrecover` wouldn't work well for contracts, so instead EIP1271 should be used i.e the `isValidSignature()`, since they use the magic value of EIP712 and not the classic v,r,s method of signing.

## Recommendation

***: Consider adding contract signature support by implementing a recovery via the suggested `isValidSignature()` function of the EIP1271 and comparing the recovered value against the `MAGIC_VALUE`

## Client Response

client response : Acknowledged.

# NEO-4:Use of ecrecover is Susceptible to Signature Malleability

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Signature Forgery or Replay | Low | Fixed | *** |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L86
- code/contracts/bridge/BridgeImpl.sol#L324

```
86: if (!management.verifyValidatorSignatures(_depositRoot, _signatures))
```

```
324: !management.verifyValidatorSignatures(
```

- code/contracts/management/BridgeManagementImpl.sol#L29-L63
- code/contracts/management/BridgeManagementImpl.sol#L29-L64
- code/contracts/management/BridgeManagementImpl.sol#L44-L47
- code/contracts/management/BridgeManagementImpl.sol#L45-L46
- code/contracts/management/BridgeManagementImpl.sol#L46

```
29: function verifyValidatorSignatures(
30:         bytes32 _newDepositRoot,
31:         BridgeLib.Signature[] calldata _signatures
32:     ) external view returns (bool) {
33:         uint256 threshold = validatorThreshold;
34:         if (_signatures.length != threshold) {
35:             return false;
36:         }
37:         bytes32 signedRootMsg = keccak256(
38:             abi.encodePacked(
39:                 "\x19Ethereum Signed Message:\n32",
40:                 keccak256(abi.encodePacked(_newDepositRoot))
41:             )
42:         );
43:         address[] memory recovered = new address[](threshold);
44:         for (uint i = 0; i < threshold; i++) {
45:             BridgeLib.Signature calldata sig = _signatures[i];
46:             recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
47:         }
48:         // check if all recovered addresses are in the validator set
```

```
        uint covered = 0;
        uint n = 0;
        uint j;
        uint validatorsLength = validators.length;
        for (uint i = 0; i < threshold; i++) {
            for (j = n; j < validatorsLength; j++) {
                if (recovered[i] == validators[j]) {
                    covered++;
                    break;
                }
            }
            n = j + 1;
        }
        return covered == threshold;
    }
```

```
function verifyValidatorSignatures(
        bytes32 _newDepositRoot,
        BridgeLib.Signature[] calldata _signatures
    ) external view returns (bool) {
        uint256 threshold = validatorThreshold;
        if (_signatures.length != threshold) {
            return false;
        }
        bytes32 signedRootMsg = keccak256(
            abi.encodePacked(
                "\x19Ethereum Signed Message:\n32",
                keccak256(abi.encodePacked(_newDepositRoot))
            )
        );
        address[] memory recovered = new address[](threshold);
        for (uint i = 0; i < threshold; i++) {
            BridgeLib.Signature calldata sig = _signatures[i];
            recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
        }
        // check if all recovered addresses are in the validator set
```

```
        uint covered = 0;
        uint n = 0;
        uint j;
        uint validatorsLength = validators.length;
        for (uint i = 0; i < threshold; i++) {
            for (j = n; j < validatorsLength; j++) {
                if (recovered[i] == validators[j]) {
                    covered++;
                    break;
                }
            }
            n = j + 1;
        }
        return covered == threshold;
    }
```

```
for (uint i = 0; i < threshold; i++) {
            BridgeLib.Signature calldata sig = _signatures[i];
            recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
        }
```

```
45: BridgeLib.Signature calldata sig = _signatures[i];
46:             recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
```

```
46: recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
```

## Description

***: The contract seems to be using Solidity `ecrecover()` function directly to verify the given signatures. However, the `ecrecover()` EVM opcode allows malleable (non-unique) signatures and thus is susceptible to replay attacks.
More Details

1. https://swcregistry.io/docs/SWC-117/
2. https://www.youtube.com/watch?v=9ZKY8DoVCkI

***: The `verifyValidatorSignatures` function in the `BridgeManagementImpl` contract uses the `ecrecover()` function to recover the address from a signature. However, `ecrecover` is susceptible to signature malleability, which can allow an attacker to create different signatures that recover the same address. This can potentially be exploited to replay attacks and bypass signature verification.
***:

## Vulnerability Detail

`verifyValidatorSignatures` ecrecover is susceptible to signature malleability which allows attacker to have a valid signature without the target user's private key.

```
function verifyValidatorSignatures(
    bytes32 _newDepositRoot,
    BridgeLib.Signature[] calldata _signatures
) external view returns (bool) {
    uint256 threshold = validatorThreshold;
    if (_signatures.length != threshold) {
        return false;
    }
    bytes32 signedRootMsg = keccak256(
        abi.encodePacked(
            "/x19Ethereum Signed Message:/n32",
            keccak256(abi.encodePacked(_newDepositRoot))
        )
    );
    address[] memory recovered = new address[](threshold);
    for (uint i = 0; i < threshold; i++) {
        BridgeLib.Signature calldata sig = _signatures[i];
        recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
    }
    // check if all recovered addresses are in the validator set
    uint covered = 0;
    uint n = 0;
    uint j;
    uint validatorsLength = validators.length;
    for (uint i = 0; i < threshold; i++) {
        for (j = n; j < validatorsLength; j++) {
            if (recovered[i] == validators[j]) {
                covered++;
                break;
            }
        }
        n = j + 1;
    }
    return covered == threshold;
}
```

In Ethereum, the ecrecover function is used to verify signatures. It is a pre-compiled contract that performs public key recovery for elliptic curve cryptography. This means it can recover a public key (address) from a given signature.
The ecrecover function, by itself, doesn't provide protection against signature malleability attacks, see reference
ECDSA signatures are malleable, meaning they can be altered without invalidating the signature.
For every valid signature (r, s), there exists another valid signature with a different s value, let's call it s. In Ethereum, the v value can also be changed accordingly. This is because the elliptic curve is symmetric, and for each point (x, y) on the curve, there's a corresponding point (x, -y) that maintains the same relationship.
An attacker can exploit this vulnerability by altering the v, r, and s values of a signature and still pass the ecrecover verification.The function takes four arguments: the hash of the message (hash), the recovery identifier

(v), and two values r and s that make up the signature.

## Impact

The signature malleability vulnerability can lead to:
Unauthorized actions or transactions in a smart contract.
Tampering with signature data to avoid security checks.
Pretending to be someone else or falsely attributing actions to others.
Due to the symmetric structure of elliptic curves, for every set of v, r, s there is another different set of v, r, s that also has the same relationship.
This results in two valid signatures and allows malicious actors to compute a valid signature without the target user's private key.
***:

## Vulnerability Detail

The solidity function ecrecover is used, however the error result of 0 is not checked for. See documentation
"recover the address associated with the public key from elliptic curve signature or return zero on error."
If invalid input parameters to `ecrecover` method are provided in `verifyValidatorSignatures`, will then result 0.
The ecrecover function returns the signer of a message, given the correct parameters v, r, and s.
In event of an invalid combination of parameters, the ecrecover function will return address(0) . If misapplied, the results could make it appear that a valid signature of the address(0) was produced.

```
function verifyValidatorSignatures(
        bytes32 _newDepositRoot,
        BridgeLib.Signature[] calldata _signatures
    ) external view returns (bool) {
        uint256 threshold = validatorThreshold;
        if (_signatures.length != threshold) {
            return false;
        }
        bytes32 signedRootMsg = keccak256(
            abi.encodePacked(
                "/x19Ethereum Signed Message:/n32",
                keccak256(abi.encodePacked(_newDepositRoot))
            )
        );
        address[] memory recovered = new address[](threshold);
        for (uint i = 0; i < threshold; i++) {
            BridgeLib.Signature calldata sig = _signatures[i];
            recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
        }
        // check if all recovered addresses are in the validator set
        uint covered = 0;
        uint n = 0;
        uint j;
        uint validatorsLength = validators.length;
        for (uint i = 0; i < threshold; i++) {
            for (j = n; j < validatorsLength; j++) {
                if (recovered[i] == validators[j]) {
                    covered++;
                    break;
                }
            }
            n = j + 1;
        }
        return covered == threshold;
    }
```

## Impact

`ecrecover()` returns zero address if the signature provided is invalid. If the signer provided is also zero, then all incorrect signatures will be allowed.
***: The `verifyValidatorSignatures` function is using EVM's ecrecover directly:

```
for (uint i = 0; i < threshold; i++) {
            BridgeLib.Signature calldata sig = _signatures[i];
            recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
        }
```

Currently there is no check for these signatures. It is better to check whether the length of a signature is 65 and whether `s` data of a signature is within valid bounds to avoid signature malleability. You can see more details about the signature check in the ECDSA.sol: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.9.3/contracts/utils/cryptography/ECDSA.sol

***: `BridgeImplementationImpl.verifyValidatorSignatures()` utilizes Solidity built-in `ecrecover()` function for verfying validator signature. The problem is that `ecrecover()` does not protect the well-known signature malleability attack.

### Impact

Malicious validator can double the amount of valid signatures that they can provide.
For example, say there are 7 validators, threshold is 4, and 2 of the validators are malicious. Without this signature malleability bug, they can only provide 2 valid signatures for a deposit root, which does not meet threshold.
However, with this bug, each of the malicious validator can forge another valid signature, thus they will have 4 valid signatures, which meets the threshold.
With those 4 valid signatures, they can bypass the `management.verifyValidatorSignatures()` check and include a fake deposit root in the system, causing financial loss.

### Math behind ECDSA signature malleability attack

Given an ordered pair (r, s), it is possible to forge a valid signature with the following formula:

```
bytes32 s2 = bytes32(uint256(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141) - uint256
(s));
```

where `0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141` is the curve order for secp256k1: the underlying elliptic curve for ECDSA algorithm.
This attack is possible because ECDSA algorithm only verifies the x-coordinate of a random elliptic curve generated by the signer. That means y-coordinate is unchecked:

```
z/s    *G + r/s    *P = (r,y)
z/(n-s)*G + r/(n-s)*P = (r, p-y)
```

- z: 32-byte hash of the message that we're signing
- s: (z+re)/k
- G: generator point of secp256k1
- r: x-coordinate of R = kG
- P: public key (P = eG, e is private key)
- y: y-coordinate of R = kG
- n: curve_order (modulus of scalar field)
- p: field_modulus (modulus of base field)

Since ECDSA algorithm only verifies if x-coordinate of R matches, we can find a symmetric elliptic curve point (r, p-y) to trick the verifier. The easiest way to do this is just to compute n - s.
For more info about the math behind this attack, check out this article.
***: The ecrecover function is used to recover the address associated with a public key from an elliptic curve signature. However, if an error occurs during this process, ecrecover returns the zero address (address(0)). The current implementation of verifyValidatorSignatures does not check for this condition
***:
One of the most critical aspects to note about **ecrecover** is its vulnerability to malleable signatures. This means that a valid signature can be transformed into a different valid signature without needing access to the private key. The potential for signature malleability introduces challenges when relying on signatures for uniqueness or identification.

```
for (uint i = 0; i < threshold; i++) {
          BridgeLib.Signature calldata sig = _signatures[i];
          recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
      }
```

Ref
***:
Take a look at https://github.com/Secure3Audit/code_NeoX_Bridge_Contract/blob/f3764f6770d85e9438997630adac73f5314afbf4/code/contracts/management/BridgeManagementImpl.sol#L29-L64

```
function verifyValidatorSignatures(
    bytes32 _newDepositRoot,
    BridgeLib.Signature[] calldata _signatures
) external view returns (bool) {
    uint256 threshold = validatorThreshold;
    if (_signatures.length != threshold) {
        return false;
    }
    bytes32 signedRootMsg = keccak256(
        abi.encodePacked(
            "\x19Ethereum Signed Message:\n32",
            keccak256(abi.encodePacked(_newDepositRoot))
        )
    );
    address[] memory recovered = new address[](threshold);
    for (uint i = 0; i < threshold; i++) {
        BridgeLib.Signature calldata sig = _signatures[i];
        recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
    }
    // check if all recovered addresses are in the validator set
    uint covered = 0;
    uint n = 0;
    uint j;
    uint validatorsLength = validators.length;
    for (uint i = 0; i < threshold; i++) {
        for (j = n; j < validatorsLength; j++) {
            if (recovered[i] == validators[j]) {
                covered++;
                break;
            }
        }
        n = j + 1;
    }
    return covered == threshold;
}
```

This function is used to verify the validator signatures, it uses the classic `ecrecover`, issue however is that it doesnt take into account the malleable nature of this signature checker, this is quite a common issue and more can be read here: https://detectors.auditbase.com/signature-malleability-of-evms-ecrecover.
TLDR of the issue is that The `ecrecover` function can return a valid address from a malformed signature (one that has been subtly altered but still valid under the secp256k1 curve used in Ethereum), i.e in our case here the `sig.s` if not enforced to either be >/ < `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0` would lead to two acceptable value for this, so is the case with `sig.v`, if not enforced to either of 27 || 28, would allow for malleable use.
That means this bug case allows for the check below to be bypassed while depositing tokens since making some changes to the `s` and `v` value allows the bypass easily, https://github.com/Secure3Audit/code_NeoX_Bridge_C-

ontract/blob/f3764f6770d85e9438997630adac73f5314afbf4/code/contracts/bridge/BridgeImpl.sol#L323-L328

```
        if (
            !management.verifyValidatorSignatures(
                _tokenDepositRoot,
                _signatures
            )
        ) revert InvalidValidatorSignatures();
```

***:** ### Summary
The `verifyValidatorSignatures()` function in the `BridgeImpl::depositGas()` and `BridgeImpl::depositToken()` function uses `ecrecover` for signature verification, which is vulnerable to signature malleability attacks.

## Impact

Potential signature malleability can allow attackers to submit alternative but valid signatures, potentially leading to unauthorized or fraudulent transactions.

# Code

```
address[] memory recovered = new address[](threshold);
        for (uint i = 0; i < threshold; i++) {
            BridgeLib.Signature calldata sig = _signatures[i];
            recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);// @audit sig malleability
        }
```

# Recommendation

***:** Use the `recover()` function from OpenZeppelin's ECDSA library for signature verification.
Reference - https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/ECDSA.sol
***:** To mitigate the risk of signature malleability, you can use the OpenZeppelin library's ECDSA functions, which provide safer alternatives for signature verification.
***:** Consider using OpenZeppelin's ECDSA library.

```
    function verifyValidatorSignatures(
        bytes32 _newDepositRoot,
        BridgeLib.Signature[] calldata _signatures
    ) external view returns (bool) {
        uint256 threshold = validatorThreshold;
        if (_signatures.length != threshold) {
            return false;
        }
        bytes32 signedRootMsg = keccak256(
            abi.encodePacked(
                "/x19Ethereum Signed Message:/n32",
                keccak256(abi.encodePacked(_newDepositRoot))
            )
        );
        address[] memory recovered = new address[](threshold);
        for (uint i = 0; i < threshold; i++) {
            BridgeLib.Signature calldata sig = _signatures[i];
-           recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
+           recovered[i] = ECDSA.recover(signedRootMsg, sig.v, sig.r, sig.s);

        }
        // check if all recovered addresses are in the validator set
        uint covered = 0;
        uint n = 0;
        uint j;
        uint validatorsLength = validators.length;
        for (uint i = 0; i < threshold; i++) {
            for (j = n; j < validatorsLength; j++) {
                if (recovered[i] == validators[j]) {
                    covered++;
                    break;
                }
            }
            n = j + 1;
        }
        return covered == threshold;
    }
```

***: ## Recommendation
The recommendation is made for adding a require statement that ensures that the result of ecrecover != 0, so that in the future, any uses of it do not mistakenly give the impression that a valid signature of address(0) was produced.

```
 function verifyValidatorSignatures(
        bytes32 _newDepositRoot,
        BridgeLib.Signature[] calldata _signatures
    ) external view returns (bool) {
        uint256 threshold = validatorThreshold;
        if (_signatures.length != threshold) {
            return false;
        }
        bytes32 signedRootMsg = keccak256(
            abi.encodePacked(
                "/x19Ethereum Signed Message:/n32",
                keccak256(abi.encodePacked(_newDepositRoot))
            )
        );
        address[] memory recovered = new address[](threshold);
        for (uint i = 0; i < threshold; i++) {
            BridgeLib.Signature calldata sig = _signatures[i];
            recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
+           require(recovered[i] != address(0), 'address Zero');

        }
        // check if all recovered addresses are in the validator set
        uint covered = 0;
        uint n = 0;
        uint j;
        uint validatorsLength = validators.length;
        for (uint i = 0; i < threshold; i++) {
            for (j = n; j < validatorsLength; j++) {
                if (recovered[i] == validators[j]) {
                    covered++;
                    break;
                }
            }
            n = j + 1;
        }
        return covered == threshold;
    }
```

***: Consider following fix:

```
for (uint i = 0; i < threshold; i++) {
            BridgeLib.Signature calldata sig = _signatures[i];
            recovered[i] = ECDSA.recover(signedRootMsg, sig.v, sig.r, sig.s);
        }
```

***: Use OpenZeppelin ECDSA.recover() for signature verification.

***: To mitigate this issue, it is recommended to use OpenZeppelin's ECDSA.sol library, which provides better error handling for signature recovery. The library's recover function returns the recovered address and handles errors more gracefully.
***: use the latest version of the **OpenZeppelin ECDSA** library which restricts the r values to range on the positive size and hence preventing the mallebility attack
***:
Consider integrating checks similar to the below for the passed signatures:

```
        // Standard checks for signature validity
        require(uint256(s) <= 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0, "Invalid s
ignature 's' value");
        require(v == 27 || v == 28, "Invalid signature 'v' value");

        address signer = ecrecover(hash, v, r, s);
        require(signer != address(0), "Invalid signer");
```

Alternatively, use Openzeppelin's `ECDSA.recover()` instead, whose implementation is

```
    /**
     * @dev Returns the address that signed a hashed message (`hash`) with
     * `signature`. This address can then be used for verification purposes.
     *
     * The `ecrecover` EVM opcode allows for malleable (non-unique) signatures:
     * this function rejects them by requiring the `s` value to be in the lower
     * half order, and the `v` value to be either 27 or 28.
     *
     * IMPORTANT: `hash` _must_ be the result of a hash operation for the
     * verification to be secure: it is possible to craft signatures that
     * recover to arbitrary addresses for non-hashed data. A safe way to ensure
     * this is by receiving a hash of the original message (which may otherwise
     * be too long), and then calling {toEthSignedMessageHash} on it.
     */
    function recover(bytes32 hash, bytes memory signature) internal pure returns (address) {
        (address recovered, RecoverError error) = tryRecover(hash, signature);
        _throwError(error);
        return recovered;
    }
```

***: Add check to not allow s value more
then `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0

```
  function verifyValidatorSignatures(
        bytes32 _newDepositRoot,
        BridgeLib.Signature[] calldata _signatures
    ) external view returns (bool) {
        uint256 threshold = validatorThreshold;
        if (_signatures.length != threshold) {
            return false;
        }
        bytes32 signedRootMsg = keccak256(
            abi.encodePacked(
                "\x19Ethereum Signed Message:\n32",
                keccak256(abi.encodePacked(_newDepositRoot))
            )
        );
        address[] memory recovered = new address[](threshold);
        for (uint i = 0; i < threshold; i++) {
            BridgeLib.Signature calldata sig = _signatures[i];
++          if (uint256(sig.s) > 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) revert
();
            recovered[i] = ecrecover(signedRootMsg, sig.v, sig.r, sig.s);
        }
        // check if all recovered addresses are in the validator set
        uint covered = 0;
        uint n = 0;
        uint j;
        uint validatorsLength = validators.length;
        for (uint i = 0; i < threshold; i++) {
            for (j = n; j < validatorsLength; j++) {
                if (recovered[i] == validators[j]) {
                    covered++;
                    break;
                }
            }
            n = j + 1;
        }
        return covered == threshold;
    }
```

or
Use OpenZeppelin ecdsa
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/ECDSA.sol#L137

## Client Response

client response : Fixed.
This issue is valid and will be addressed.

This issue has been fixed in commit d799301808f21777eaceb0697ea2b2bc3aac7153.
client response : Fixed.
This issue is only valid if you're ignoring context outside the verifyValidatorSignatures() function.
However, with consideration of the validator setter function that does a zero address check, the mentioned issue is mitigated with the existing zero address check in the validator setter function.
Nevertheless, because the function itself should be safe from such external checks, I'll consider this issue valid. It has been fixed completely by fixing issue #7 where the OpenZeppelin's ECDSA.recover() function was introduced replacing the precompile ecrecover. ECDSA.recover() reverts with ECDSAInvalidSignature if an invalid signature is passed.
This issue is valid and will be addressed.
This issue has been fixed in commit d799301808f21777eaceb0697ea2b2bc3aac7153.

# NEO-5:UUPS proxy contracts lack initialize function and be initialized in constructor

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| DOS | Low | Fixed | *** |

## Code Reference

- code/contracts/bridge/BridgeStorage.sol#L20-L22

```
20: constructor(address _management) BridgeStorageV1(_management) {
21:         _disableInitializers();
22:     }
```

- code/contracts/management/BridgeManagementImpl.sol#L9
- code/contracts/management/BridgeManagementImpl.sol#L65
- code/contracts/management/BridgeManagementImpl.sol#L74
- code/contracts/management/BridgeManagementImpl.sol#L86
- code/contracts/management/BridgeManagementImpl.sol#L103
- code/contracts/management/BridgeManagementImpl.sol#L112
- code/contracts/management/BridgeManagementImpl.sol#L121

```
9: constructor(
```

```
65: function setOwner(address _owner) external onlyOwner {
```

```
74: function setRelayer(address _relayer) external onlyOwner {
```

```
86: ) external onlyOwner {
```

```
103: function setGovernor(address _governor) external onlyOwner {
```

```
112: function setSecurityGuard(address _securityGuard) external onlyOwner {
```

```
121: function setFunder(address _funder) external onlyOwner {
```

- code/contracts/management/BridgeManagementStorage.sol#L16-L32
- code/contracts/management/BridgeManagementStorage.sol#L16

```
16: constructor(
17:         address _owner,
18:         address _relayer,
19:         uint256 _validatorThreshold,
20:         address[] memory _validators,
21:         address _governor,
22:         address _securityGuard,
23:         address _funder
24:     ) {
25:         _disableInitializers();
26:         _setOwner(_owner);
27:         _setRelayer(_relayer);
28:         _setValidators(_validators, _validatorThreshold);
29:         _setGovernor(_governor);
30:         _setSecurityGuard(_securityGuard);
31:         _setFunder(_funder);
32:     }
```

```
16: constructor(
```

## Description

***: openzeeplin describes in detail the secure implementation of the uups proxy contract: uups proxy
it can't write anything in implication contract memory:

```
constructor() {
    _disableInitializers();
}
```

and it need an `initilize` function to call and initlized in proxy contract:

```
    function initialize() public initializer {
        hasInitialValue = 42; // set initial value in initializer
    }
```

In BridgeManagementImpl and BridgeImpl contract,they all lack initialize function and initialized in constructor(bridge contract as example):

```
    constructor(address _management) BridgeStorage(_management) {}
```

```
    constructor(address _management) BridgeStorageV1(_management) {
        _disableInitializers();
    }
```

```
    constructor(address _management) {
        require(management != address(0));
        management = IBridgeManagement(_management);
        gasBridge = StorageTypes.GasBridge({
            paused: false,
            depositState: StorageTypes.State({nonce: 0, root: 0x0}),
            withdrawalState: StorageTypes.State({nonce: 0, root: 0x0}),
            config: StorageTypes.GasConfig({
                fee: 1e17,
                minAmount: 1e18,
                maxAmount: 1e22,
                maxDeposits: 100
            })
        });
    }
```

It will cause the implementation contract to initialize (it can't to be initlized), but the proxy contract will not (it should be).The UUPS proxy contract will not work.
***:

## Vulnerability Detail

`BridgeManagementStorage` and `BridgeManagementImpl` both contract are upgradeable and use constructor against OZ guideline;
'You can use your Solidity contracts with OpenZeppelin Upgrades without any modifications, except for their constructors. Due to a requirement of the proxy-based upgradeability system, no constructors can be used in upgradeable contracts'.

## POC

```
contract BridgeManagementImpl is BridgeManagementStorage, IBridgeManagement {
    constructor(
        address _owner,
        address _relayer,
        uint8 _validatorThreshold,
        address[] memory _validators,
        address _governor,
        address _securityGuard,
        address _funder
    )
        BridgeManagementStorage(
            _owner,
            _relayer,
            _validatorThreshold,
            _validators,
            _governor,
            _securityGuard,
            _funder
        )
    {}
```

Same is case for ```BridgeManagementStorage```` using constructor in upgradebale contract.

```
contract BridgeManagementStorage is BridgeManagementStorageV1, UUPSUpgradeable {
    address public constant SELF = 0x1212100000000000000000000000000000000005;
    address public constant GOV_ADMIN =
        0x1212000000000000000000000000000000000000;

    constructor(
        address _owner,
        address _relayer,
        uint256 _validatorThreshold,
        address[] memory _validators,
        address _governor,
        address _securityGuard,
        address _funder
    ) {
        _disableInitializers();
        _setOwner(_owner);
        _setRelayer(_relayer);
        _setValidators(_validators, _validatorThreshold);
        _setGovernor(_governor);
        _setSecurityGuard(_securityGuard);
        _setFunder(_funder);
    }
```

## Impact

When a upgrade call is made the contract will lost all values set in constructor as they are stored in proxy which will be unable to finds and will halt all contract working as well loss of funds if stored in any of these contract.
***:**

## Vulnerability Detail

`BridgeManagementImpl` inherits `BridgeManagementStorageV1` which is upgradeable contract as it inherits the `UUPSUpgradeable` contract.
Various function i.e., `setOwner`, `setRelayer`, `setValidators`, `setGovernor`, `setSecurityGuard` and `setFunder` are used to set managing role for the protocol in `BridgeManagementImpl` contract which will become useless when an upgrade call is made to the implementation contract.

## POC

OZ clearly mentions that; `Due to a requirement of the proxy-based upgradeability system, no constructors can be used in upgradeable contracts`.
This is due to state variable are stored in proxies and not in implementation contract, as OZ mentions , `proxies are completely oblivious to the storage trie changes that are performed by the constructor`.
So when update call is made `onlyOwner` will be `address(0)` which can restrict setting these role.
***: Using a constructor in a **UUPS (Universal Upgradeable Proxy Standard)** contract poses significant issues because the proxy pattern used in UUPS requires the implementation contract to be initialized through a proxy, not directly. When a constructor is used, it sets the initial values only in the implementation contract, which isn't accessed during normal operations. The proxy, which delegates calls to the implementation contract, does not retain these initialized values. Consequently, any state set in the constructor is effectively lost, leading to uninitialized variables and potential security risks.
With respect to the current contract, the deployment of the contract will lead to uninitiazed owner, relayer, validator, funder and governor. There are custom functions to set this individually that needs to called to set these values.
For more detailed information about using constructors in upgradable contracts. Read here - https://docs.openzeppelin.com/learn/upgrading-smart-contracts#initialization

## Recommendation

***: add `initialize` function,and delete all initializing logic in constructor.
***: The recommendation is made to use `initialize()` function instead of setting state variables in constructor for both `BridgeManagementStorage` and `BridgeManagementImpl` contract as provided by OZ's ; ''The problem is easily solved though. Logic contracts should move the code within the constructor to a regular 'initializer' function''.
***: The recommendation is made to use `initialize()` function instead of setting state variables in constructor of upgradeable contracts.
***: Instead of using a constructor, implement an `initialize` function marked with the `initializer` modifier provided by **OpenZeppelin's Initializable** contract. This function will replace the constructor for setting up initial values.

## Client Response

client response : Fixed. This issue has been fixed in commit 15ff54a8c5060906d236114f71bfac875bfebd2c to provide a more concise development process.
This issue is partially valid.
Neo X will be a new chain and the bridge will be natively integrated. The bridge contract's deployed code will be included in the genesis file together with the initial storage allocation. This means:
the contract will not be formally "deployed" -> the constructor will have no effect, as it will never be executed and is not included in the deployed byte code
no initializer function is necessary
Having the constructor setting values is necessary for the UTs, since they are not written testing the contract via a proxy. Nevertheless, it is a confusing part in the development and could be improved. Especially in regard to potential future versions of the logic contract. In order to improve it, the UTs could be changed to test the contract via proxy moving the constructor logic to an initializer function.
Based on all above, we will change the severity of this issue to low or informational. The decision whether it will be further addressed has not been made yet.
The severity of this issue is low. The reason for this has been stated in the above comment.
Additionally to this clarification in the comment, we consider this issue having low severity because the contracts are developed to be native contracts and the constructor will not have any effect when setting its deployed byte code in the genesis file. Meaning that this issue does not have any impact or can just be considered as a warning that could remain unfixed.
Nevertheless, we're considering fixing/improving it in order to provide a more concise development process, increasing clarity that this contract is only considered to be deployed behind a proxy.
client response : Fixed. This issue has been fixed in commit 15ff54a8c5060906d236114f71bfac875bfebd2c to provide a more concise development process.
This issue is partially valid.
Neo X will be a new chain and the bridge will be natively integrated. The bridge contract’s deployed code will be included in the genesis file together with the initial storage allocation. This means:
the contract will not be formally “deployed” -> the constructor will have no effect, as it will never be executed and is not included in the deployed byte code
no initializer function is necessary
Having the constructor setting values is necessary for the UTs, since they are not written testing the contract via a proxy. Nevertheless, it is a confusing part in the development and could be improved. Especially in regard to potential future versions of the logic contract. In order to improve it, the UTs could be changed to test the contract via proxy moving the constructor logic to an initializer function.
Based on all above, we will change the severity of this issue to low or informational. The decision whether it will be further addressed has not been made yet.
The severity of this issue is low. The reason for this has been stated in the above comment.
Additionally to this clarification in the comment, we consider this issue having low severity because the contracts are developed to be native contracts and the constructor will not have any effect when setting its deployed byte code in the genesis file. Meaning that this issue does not have any impact or can just be considered as a warning that could remain unfixed.
Nevertheless, we’re considering fixing/improving it in order to provide a more concise development process, increasing clarity that this contract is only considered to be deployed behind a proxy.

# NEO-6:The value of `minAmount` should not be equal to the value of `maxAmount`.

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Low | Fixed | *** |

## Code Reference

- code/contracts/bridge/BridgeStorage.sol#L260-L276

```
260: function _setTokenMinWithdrawalAmount(
261:         address _neoXToken,
262:         uint256 _amount
263:     ) internal {
264:         if (_amount > tokenBridges[_neoXToken].config.maxAmount)
265:             revert InvalidAmount();
266:         tokenBridges[_neoXToken].config.minAmount = _amount;
267:     }
268:
269:     function _setTokenMaxWithdrawalAmount(
270:         address _neoXToken,
271:         uint256 _amount
272:     ) internal {
273:         if (_amount < tokenBridges[_neoXToken].config.minAmount)
274:             revert InvalidAmount();
275:         tokenBridges[_neoXToken].config.maxAmount = _amount;
276:     }
```

## Description

***: The `_setTokenMinWithdrawalAmount` function allows the `minAmount` to be set equal to `maxAmount`. This is problematic because typically, the minimum amount should be strictly less than the maximum amount to ensure logical consistency and prevent potential misuse or errors in the smart contract's functionality.

## Recommendation

***: Add a check to ensure that `minAmount` is strictly less than `maxAmount`. This can be done by modifying the condition in the function as follows:

```
function _setTokenMinWithdrawalAmount(address _neoXToken, uint256 _amount) internal {
    if (_amount >= tokenBridges[_neoXToken].config.maxAmount)
        revert InvalidAmount();
    tokenBridges[_neoXToken].config.minAmount = _amount;
}
```

This change ensures that the minimum withdrawal amount is always less than the maximum withdrawal amount, maintaining the logical integrity of the contract.

## Client Response

client response : Fixed. This issue has been fixed in commit f056c538e78f4b0bea52482459c350a07de7ccf1.

# NEO-7:Inconsistent balance on transfer-on-fee or deflationary tokens

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Low | Fixed | *** |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L442-L505

```
442: */
443:     function withdrawToken(
444:         address _neoXToken,
445:         address _to,
446:         uint256 _amount
447:     )
448:         external
449:         payable
450:         override
451:         nonReentrant
452:         onlyBridgeUnpaused
453:         onlyTokenBridgeUnpaused(_neoXToken)
454:     {
455:         if (!_isRegisteredToken(_neoXToken))
456:             revert TokenBridgeNotRegistered(_neoXToken);
457:         StorageTypes.TokenConfig memory config = _getTokenConfig(_neoXToken);
458:         uint256 tokenValue = _amount;
459:         if (tokenValue < config.minAmount) revert InvalidAmount();
460:         if (tokenValue > config.maxAmount) revert InvalidAmount();
461:         if (msg.value < config.fee)
```

```
462:             revert InsufficientFee(msg.value, config.fee);
463:         _addUnclaimedRewards(msg.value);
464:
465:         // Execute the transfer of the tokens from the sender to the bridge contract.
466:         bool success = IERC20(_neoXToken).transferFrom(
467:             msg.sender,
468:             address(this),
469:             _amount
470:         );
471:         if (!success) revert TransferFailed();
472:
473:         // Compute the new root and update the token withdrawal state.
474:         StorageTypes.State memory state = _getTokenWithdrawalState(_neoXToken);
475:         uint256 newNonce = state.nonce + 1;
476:
477:         if (config.executionType == StorageTypes.ExecutionType.NEO) {
478:             if (tokenValue % 1e18 != 0) {
479:                 revert InvalidAmount();
480:             }
481:             tokenValue /= 1e18;
```

```
482:         }
483:
484:         bytes32 withdrawalHash = TokenBridgeLib._hashTokenBridgeOp(
485:             config.neoN3Token,
486:             _neoXToken,
487:             newNonce,
488:             _to,
489:             tokenValue
490:         );
491:         bytes32 newRoot = BridgeLib._computeNewRoot(state.root, withdrawalHash);
492:         _setTokenWithdrawalState(
493:             _neoXToken,
494:             StorageTypes.State({nonce: newNonce, root: newRoot})
495:         );
496:         emit TokenWithdrawal(
497:             _neoXToken,
498:             config.neoN3Token,
499:             newNonce,
500:             _to,
501:             tokenValue,
```

```
502:             msg.sender,
503:             withdrawalHash,
504:             newRoot
505:         );
```

## Description

***:

Take a look at https://github.com/Secure3Audit/code_NeoX_Bridge_Contract/blob/f3764f6770d85e9438997630adac73f5314afbf4/code/contracts/bridge/BridgeImpl.sol#L442-L505

```
function withdrawToken(
    address _neoXToken,
    address _to,
    uint256 _amount
)
    external
    payable
    override
    nonReentrant
    onlyBridgeUnpaused
    onlyTokenBridgeUnpaused(_neoXToken)
{
    if (!_isRegisteredToken(_neoXToken))
        revert TokenBridgeNotRegistered(_neoXToken);
    StorageTypes.TokenConfig memory config = _getTokenConfig(_neoXToken);
    uint256 tokenValue = _amount;
    if (tokenValue < config.minAmount) revert InvalidAmount();
    if (tokenValue > config.maxAmount) revert InvalidAmount();
    if (msg.value < config.fee)
        revert InsufficientFee(msg.value, config.fee);
    _addUnclaimedRewards(msg.value);

    // Execute the transfer of the tokens from the sender to the bridge contract.
    //@audit
    bool success = IERC20(_neoXToken).transferFrom(
        msg.sender,
        address(this),
        _amount
    );
    if (!success) revert TransferFailed();

    // Compute the new root and update the token withdrawal state.
    StorageTypes.State memory state = _getTokenWithdrawalState(_neoXToken);
    uint256 newNonce = state.nonce + 1;

    if (config.executionType == StorageTypes.ExecutionType.NEO) {
        if (tokenValue % 1e18 != 0) {
            revert InvalidAmount();
        }
        tokenValue /= 1e18;
    }

    bytes32 withdrawalHash = TokenBridgeLib._hashTokenBridgeOp(
        config.neoN3Token,
        _neoXToken,
        newNonce,
        _to,
```

This function is used to withdraw tokens to Neo N3 and it requires that the sender has approved the provided amount to the bridge contract, issue however is that when transferring the execution uses the classic `IERC20.transferFrom()`, however for popular tokens that deduct fees on their transfers, this then causes protocol to not only inflate the amount of tokens received, but some invariants could also be broken.
POC for broken invariant:

- Here we are to prove a user can deposit < config.minAmount.
- config.minAmount for the token is set as 10e18.
- User calls on `withdraw` with 106e17.
- The token's fee for transfers is 10%
- User ends up depositing ~ 95e17 which is < minAmount = 10e18

As shown by the minimalistic POC, not only is the accounting flawed, but users could as well even break an invariant.

## Recommendation

***:
Introduce a balance before and after the transfer check, then the difference between these balances should be the real deposited amount, i.e `tokenValue` and then the checks should be done on this amount not to be < config.minAmount.

## Client Response

client response : Fixed. This issue has been fixed in commit 1c3428762804c6ba0294f042be22e60c29634383.

# NEO-8:msg.value exceeding the required fee is lost by the user

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Informational | Fixed | *** |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L176
- code/contracts/bridge/BridgeImpl.sol#L462
- code/contracts/bridge/BridgeImpl.sol#L463

```
176: _addUnclaimedRewards(config.fee);
```

```
462: revert InsufficientFee(msg.value, config.fee);
```

```
463: _addUnclaimedRewards(msg.value);
```

- code/contracts/bridge/BridgeStorage.sol#L116-118

```
116: function _addUnclaimedRewards(uint256 _amount) internal {
117:         unclaimedRewards += _amount;
118:     }
```

## Description

***: ### Summary
The `withdrawToken()` function in the BridgeImpl.sol does not return excess `msg.value` to the sender; it only checks if the amount is sufficient and then adds the entire value to unclaimed rewards.

### Impact

Users may unintentionally lose funds, as any excess `msg.value` beyond the required fee is not refunded.

### Code

```
if (msg.value < config.fee)
    revert InsufficientFee(msg.value, config.fee);
_addUnclaimedRewards(msg.value);
```

***: The variable `unclaimedRewards` will only be added a new value via the call of the function `_addUnclaimedRewards()`. And the function `_addUnclaimedRewards()` is called in the functions `withdrawGas()` and `withdrawToken()`.
In the function `withdrawGas()`, the call of `_addUnclaimedRewards()` will add `config.fee` to the `unclaimedRewards`.
But in the function `withdrawToken()`, the call of `_addUnclaimedRewards()` will add `msg.value` to the `unclaimedRewar`

`s`.
Besides, the value of `unclaimedRewards` is never used in other functions.
As a result, the meaning of the variable `unclaimedRewards` is not clear and is not useful.
***: When users calls the `withdrawToken` function, they have to send the fee in form of `msg.value`. The function only checks the if the sent `msg.value` is less than fee, and if it's not, adds `msg.value` NOT the fee to the unclaimed rewards which leads to loss of funds for any users who sends more that the fee. This also goes against the implementation in the `withdrawGas` function.

```
    function withdrawToken(
        address _neoXToken,
        address _to,
        uint256 _amount
    )
......
        if (msg.value < config.fee)
            revert InsufficientFee(msg.value, config.fee);
        _addUnclaimedRewards(msg.value); //@audit
```

## Recommendation

***: After deducting the required fee, return any excess `msg.value` to the sender to prevent unintended loss of funds.

```
    function withdrawToken(
        address _neoXToken,
        address _to,
        uint256 _amount
    )
        external
        payable
        override
        nonReentrant
        onlyBridgeUnpaused
        onlyTokenBridgeUnpaused(_neoXToken)
    {
        if (!_isRegisteredToken(_neoXToken))
            revert TokenBridgeNotRegistered(_neoXToken);
        StorageTypes.TokenConfig memory config = _getTokenConfig(_neoXToken);
        uint256 tokenValue = _amount;
        if (tokenValue < config.minAmount) revert InvalidAmount();
        if (tokenValue > config.maxAmount) revert InvalidAmount();
        if (msg.value < config.fee)
            revert InsufficientFee(msg.value, config.fee);
        _addUnclaimedRewards(msg.value);// @audit return access fee
++      if (msg.value > config.fee) {
++          payable(msg.sender).transfer(msg.value - config.fee);
++      }
        // Execute the transfer of the tokens from the sender to the bridge contract.
        bool success = IERC20(_neoXToken).transferFrom(
            msg.sender,
            address(this),
            _amount
        );
```

***: Recommend confirming the logic of `unclaimedRewards` and confirm the value added to the variable in the functions `withdrawGas()` and `withdrawToken()`.
***: Consider checking that `msg.value` == `config.fee` instead and adding config.fee to unclaimed rewards., or sending back excess tokens to `msg.sender`,

```
        if (msg.value != config.fee)
            revert InsufficientFee(msg.value, config.fee);
        _addUnclaimedRewards(config.fee);
```

## Client Response

client response : Fixed. The fix has been done in commit 0041b5c789e81ab51654e915ce0d29827835ff18. Further, the value unclaimedRewards is clear. Its distribution is not yet clear, hence this value just keeps track of what amount of fees have been collected so far to provide more transparent feedback about it to bridge operators and users (instead of some off-chain computation once there's a distributeFee function in the future).

client response : Fixed. The fix has been done in commit 0041b5c789e81ab51654e915ce0d29827835ff18. Further, the value unclaimedRewards is clear. Its distribution is not yet clear, hence this value just keeps track of what amount of fees have been collected so far to provide more transparent feedback about it to bridge operators and users (instead of some off-chain computation once there's a distributeFee function in the future).

# NEO-9:Validator threshold can be set to 1 and malicious validator can forge Merkle root

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Privilege Related | Informational | Fixed | *** |

## Code Reference

- code/contracts/management/BridgeManagementStorage.sol#L53-L54

```
53: if (_threshold == 0 || _threshold > validatorsLength)
54:             revert InvalidValidatorThreshold();
```

## Description

***: In `_setValidators()`, it checks that validator threshold can't be 0 or exceed the length of `_validators` array. However, it is possible to set threshold to one, which means any single validator can make decision. That introduces centralization risk when any of the validators is malicious.
In trust model definitions:

- Semi-trusted: trust that at least a threshold number (for example 2/3 majority) are honest
- Fully trusted: trust that a single party is honest

If threshold can be set to 1, we can see that the trust model was supposed to be semi-trusted, but it downgrades to fully trusted when threshold=1.
If threshold is set to 1 and that validator is malicious, the result is devastating. For example, malicious validator can forge a fake deposit root offchain and sign it, then send merkle root and signature to relayer. Relayer will call `BridgeImpl.depositGas()` and that fake merkle root will be treated as valid. In the worst case malicious validator can mint huge amount of GAS tokens on the other chain, breaking the economic model of the bridge.

## Recommendation

***: Set a non-zero lower bound for validator threshold, such as 2 or higher.

## Client Response

client response : Fixed. This has been fixed in commit 8b5161a13b491133ff187ae791f2f8140fa22075.

# NEO-10:Unused error: `TokenWithdrawalFailed`

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Informational | Fixed | *** |

## Code Reference

- code/contracts/bridge/BridgeStorage.sol#L44

```
44: error TokenWithdrawalFailed();
```

## Description

***: The error `TokenWithdrawalFailed` is declared but not used

## Recommendation

***: Remove this error:

```
@@ -41,7 +41,6 @@ contract BridgeStorage is BridgeStorageV1, UUPSUpgradeable {
     error TokenBridgePaused(address neoXToken);
     error TokenBridgeUnpaused(address neoXToken);
     error TokenBridgeNotRegistered(address neoXToken);
-    error TokenWithdrawalFailed();
     error TransferFailed();
```

## Client Response

client response : Fixed. This has been fixed in commit 78a0abd4d06b6f3e8e74015ddd69de3cc2c16304.

# NEO-11:Unlocked Pragma Version

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Code Style | Informational | Fixed | *** |

## Code Reference

- code/contracts/management/BridgeManagementImpl.sol#L2

```
2: pragma solidity ^0.8.24;
```

## Description

***: Solidity files in packages have a pragma version `^0.8.24`. The caret `(^)` points to unlocked pragma, meaning the compiler will use the specified version or above.

## Recommendation

***: It's good practice to use specific solidity versions to know compiler bug fixes and optimisations were enabled at the time of compiling the contracts.

## Client Response

client response : Fixed. This has been fixed in commit e1d89c2771af29321daf29b4d1748727e804b2ca.

# NEO-12:Threshold Condition in verifyValidatorSignatures Function

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Privilege Related | Informational | Acknowledged | *** |

## Code Reference

- code/contracts/management/BridgeManagementImpl.sol#L34

```
34: if (_signatures.length != threshold) {
```

## Description

***: The `verifyValidatorSignatures` function requires that the amount of provided signatures is exactly equal to the threshold. Since, a majority of owner can be potentially interested in execution, it makes more sense to allow as many signatures as possible regardless of the threshold, provided the threshold is met. The current implementation will retrun false if more than threshold signature is passed.

```
    function verifyValidatorSignatures(
        bytes32 _newDepositRoot,
        BridgeLib.Signature[] calldata _signatures
    ) external view returns (bool) {
        uint256 threshold = validatorThreshold;
        if (_signatures.length != threshold) {
            return false;
        }
        bytes32 signedRootMsg = keccak256(
            abi.encodePacked(
                "\x19Ethereum Signed Message:\n32",
                keccak256(abi.encodePacked(_newDepositRoot))
            )
        );
 .......
```

***: The current implementation of `verifyValidatorSignatures` includes the following condition:

```
if (_signatures.length != threshold) {
    return false;
}
```

This condition ensures that the number of provided signatures matches the required threshold exactly. However, in scenarios where more than the threshold number of signatures are provided, the function should still proceed to verify the signatures as long as at least the threshold number of valid signatures are present. This ensures flexibility and robustness in the signature verification process.

## Recommendation

***: Consider using the => operator instead.

```
    if (_signatures.length < threshold) {
        return false;
    }
```

***: To address this issue, the function should be modified to allow more than the threshold number of signatures. The condition should be updated to check if the number of signatures is less than the threshold, and the function should count the number of valid signatures until the threshold is met.

```
    if (_signatures.length < threshold) {
        return false;
    }
```

## Client Response

client response : Acknowledged. This issue will be acknowledged.
We might fix this at a later stage.

# NEO-13:Support add, remove and replace functionality for validators

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Code Style | Informational | Acknowledged | *** |

## Code Reference

- code/contracts/management/BridgeManagementStorage.sol#L47-L67

```
47: function _setValidators(
48:         address[] memory _validators,
49:         uint256 _threshold
50:     ) internal {
51:         uint256 validatorsLength = _validators.length;
52:         if (validatorsLength == 0) revert InvalidValidatorArray();
53:         if (_threshold == 0 || _threshold > validatorsLength)
54:             revert InvalidValidatorThreshold();
55:         for (uint256 i = 0; i < validatorsLength; i++) {
56:             if (_validators[i] == address(0)) revert InvalidAddress();
57:         }
58:         if (ManagementLib._hasDuplicates(_validators))
59:             revert InvalidValidatorArray();
60:
61:         delete validators;
62:         for (uint256 i = 0; i < validatorsLength; i++) {
63:             validators.push(_validators[i]);
64:         }
65:         validatorThreshold = _threshold;
66:     }
```

## Description

***:

Take a look at https://github.com/Secure3Audit/code_NeoX_Bridge_Contract/blob/f3764f6770d85e9438997630adac73f5314afbf4/code/contracts/management/BridgeManagementStorage.sol#L47-L67

```
function _setValidators(
    address[] memory _validators,
    uint256 _threshold
) internal {
    uint256 validatorsLength = _validators.length;
    if (validatorsLength == 0) revert InvalidValidatorArray();
    if (_threshold == 0 || _threshold > validatorsLength)
        revert InvalidValidatorThreshold();
    for (uint256 i = 0; i < validatorsLength; i++) {
        if (_validators[i] == address(0)) revert InvalidAddress();
    }
    if (ManagementLib._hasDuplicates(_validators))
        revert InvalidValidatorArray();

    delete validators;
    for (uint256 i = 0; i < validatorsLength; i++) {
        validators.push(_validators[i]);
    }
    validatorThreshold = _threshold;
}
```

This function eventually gets called from the bridge implementation and is used to set a new set of validators. Issue however is that this functionality requires that with every call, the whole validator set must be removed and then replaced.
However common validator/threshold logic we see that, most times there is a case to either add one more validator or remove one from the set and is being suspected of malicious activity.
Taking the actions above would actually not be that possible, considering the owner has to call the function below with a whole new set.

## Recommendation

***: Consider integrating an addValidator() & removeValidator functionalities, these functions should then include threshold checks to ensure the protocol doesn't get set in a state where threshold > validators

## Client Response

client response : Acknowledged. Inaccurate title. A better title for this would be Support add, remove and replace functionality for validators.
This issue will be acknowledged.
We might fix this in the future.

# NEO-14:Solidity version 0.8.20+ may not work on other chains due to `PUSH0`

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Language Specific | Informational | Acknowledged | *** |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L2

```
2: pragma solidity ^0.8.24;
```

- code/contracts/management/BridgeManagementImpl.sol#L2

```
2: pragma solidity ^0.8.24;
```

## Description

***: The compiler for Solidity 0.8.20 switches the default target EVM version to Shanghai, which includes the new `PUSH0` op code. This op code may not yet be implemented on all L2s, so deployment on these chains will fail. To work around this issue, use an earlier EVM version. While the project itself may or may not compile with `0.8.20`, other projects with which it integrates, or which extend this project may, and those projects will have problems deploying these contracts/libraries.

## Recommendation

***: Consider using `0.8.19` if Neo X does not support `PUSH0`.

## Client Response

client response : Acknowledged. This issue will be acknowledged.
Unless this is required in the future, we will not fix this.

# NEO-15:Signature replay attack is possible for BridgeManagementImpl.verifyValidatorSignatures()

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Signature Forgery or Replay | Informational | Acknowledged | *** |

## Code Reference

- code/contracts/management/BridgeManagementImpl.sol#L37-L42

```
37: bytes32 signedRootMsg = keccak256(
38:             abi.encodePacked(
39:                 "\x19Ethereum Signed Message:\n32",
40:                 keccak256(abi.encodePacked(_newDepositRoot))
41:             )
42:         );
```

## Description

***: The computation of signedRootMsg does not include chaindId and nonce:

```
        bytes32 signedRootMsg = keccak256(
            abi.encodePacked(
                "\x19Ethereum Signed Message:\n32",
                keccak256(abi.encodePacked(_newDepositRoot))
            )
        );
```

A malicious validator can observe signatures and replay them.

## Recommendation

***: Add chainId and nonce when computing the message hash.

## Client Response

client response : Acknowledged. This issue will be acknowledged.
We will fix this issue at a later stage.

# NEO-16:Prevent underflow and provide clear feedback for insufficient fee

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Privilege Related | Informational | Fixed | *** |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L161-L199
- code/contracts/bridge/BridgeImpl.sol#L201-L204
- code/contracts/bridge/BridgeImpl.sol#L461

```
161: /**
162:      * @notice Withdraw Gas to provided address on Neo N3. The provided amount of Gas must have a precisi
on of maximal 8 decimal points due to the GAS token on Neo N3 having 8 decimals.
163:      * @dev When invoking this function provide the amount of Gas to withdraw to Neo N3 as msg.value.
164:      * @param _to the address to which the Gas should be sent on Neo N3.
165:      */
166:     function withdrawGas(
167:         address _to
168:     ) external payable onlyBridgeUnpaused onlyGasBridgeUnpaused {
169:         if (_to == address(0)) revert InvalidAddress();
170:         if ((msg.value % 1e10) != 0) revert InvalidAmount();
171:         StorageTypes.GasConfig memory config = _getGasBridgeConfig();
172:         StorageTypes.State memory state = _getGasBridgeWithdrawalState();
173:         uint256 actualWithdrawalAmount = msg.value - config.fee;
174:         if (actualWithdrawalAmount < config.minAmount) revert InvalidAmount();
175:         if (actualWithdrawalAmount > config.maxAmount) revert InvalidAmount();
176:         _addUnclaimedRewards(config.fee);
177:
178:         uint256 amountForHashing = GasBridgeLib._removeTenDecimals(
179:             actualWithdrawalAmount
180:         );
```

```
181:         uint256 newNonce = state.nonce + 1;
182:         bytes32 withdrawalHash = GasBridgeLib._hashGasBrideOp(
183:             newNonce,
184:             _to,
185:             amountForHashing
186:         );
187:         bytes32 newRoot = BridgeLib._computeNewRoot(state.root, withdrawalHash);
188:         _setGasBridgeWithdrawalState(
189:             StorageTypes.State({nonce: newNonce, root: newRoot})
190:         );
191:         emit GasWithdrawal(
192:             newNonce,
193:             _to,
194:             amountForHashing,
195:             msg.sender,
196:             withdrawalHash,
197:             newRoot
198:         );
199:     }
```

```
201: function setGasWithdrawalFee(uint256 _fee) external onlyGovernor {
202:         _setGasWithdrawalFee(_fee);
203:         emit GasWithdrawalFeeChange(_fee);
204:     }
```

```
461: if (msg.value < config.fee)
```

## Description

***: ## Summary
The `setGasWithdrawalFee` function allows setting the gas withdrawal fee without any upper limit, as long as it's a multiple of 1e10. This gives the `onlyGovernor` the power to set excessively high fees.

## Impact

- A malicious admin could set an extremely high fee, extracting large amounts from users during gas withdrawals.
- Excessive fees could disrupt normal operations and deter users from using the platform.
- Users may get charged more fees than they agreed upon initially.

## Proof of concept

- Initially, the `admin` sets the gas withdrawal fee to `1e10` by calling `setGasWithdrawalFee`.

  ```
  admin.setGasWithdrawalFee(1e10);
  ```

- Later, the `admin` decides to increase the fee and updates it to `30e10`.

  ```
  admin.setGasWithdrawalFee(30e10);
  ```

- Users, unaware of the new higher fee, call `withdrawGas` expecting the original fee of `1e10`. However, they are now charged `30e10`, significantly more than they expected or agreed to.

  ```
  user.withdrawGas{value: expectedAmount}(neoAddress);
  ```

- Users face higher fees without prior notice, leading to potential financial losses and trust issues.
- Users' transactions may fail or become more expensive, causing frustration and possible service disruption.

## Recommendation

***: - Cap the fee at a reasonable upper bound aligned with expected operational costs to prevent setting excessively high fees.

- Ensure users are promptly informed of any fee changes. This transparency allows users to be aware of the new fee structure before they initiate the `withdrawGas` function.

## Client Response

client response : Fixed. This has been fixed in commit c4e7ad86ae2dbaebfcfab303cd1fa18f5f9f5ac1.

# NEO-17:Ownership change should use two-step process

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Privilege Related | Informational | Fixed | *** |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L145-L160

```
145: function claimGas(
146:         uint256 _nonce
147:     ) external onlyBridgeUnpaused onlyGasBridgeUnpaused nonReentrant {
148:         StorageTypes.Claimable memory claimable = _getGasClaimable(_nonce);
149:         uint256 amount = claimable.amount;
150:         address to = claimable.to;
151:         if (amount == 0) revert NonexistentClaimable();
152:         if (to == address(0)) revert NonexistentClaimable();
153:
154:         _deleteGasClaimable(_nonce);
155:         uint256 sendValue = GasBridgeLib._addTenDecimals(amount);
156:         (bool success, ) = to.call{value: sendValue}("");
157:         if (!success) revert TransferFailed();
158:         emit GasClaim(_nonce, to, amount);
159:     }
```

- code/contracts/management/BridgeManagementImpl.sol#L65-L68
- code/contracts/management/BridgeManagementImpl.sol#L65-68

```
65: function setOwner(address _owner) external onlyOwner {
66:         _setOwner(_owner);
67:         emit OwnerChange(_owner);
68:     }
```

```
65: function setOwner(address _owner) external onlyOwner {
66:         _setOwner(_owner);
67:         emit OwnerChange(_owner);
68:     }
```

- code/contracts/management/BridgeManagementStorage.sol#L38-L45
- code/contracts/management/BridgeManagementStorage.sol#L43-L45

```
38: modifier onlyOwner() {
39:         require(msg.sender == owner, "not owner");
40:         _;
41:     }
42:
43:     function _setOwner(address _owner) internal {
44:         owner = _owner;
45:     }
```

```
43: function _setOwner(address _owner) internal {
44:         owner = _owner;
45:     }
```

## Description

***: ## Summary
The `BridgeManagementImpl` currently uses a one-step ownership transfer process, which poses a risk of setting the owner to an invalid address (e.g the zero address). This could disable all `onlyOwner` functions, including `setRelayer`, `setValidators`, `setGovernor`, `setSecurityGuard`, and `setFunder`.

## Impact

- Setting the owner to an invalid address will disable all critical `onlyOwner` functions, impacting contract governance and operations.
- Immediate ownership transfer can lead to unauthorized control if the owner's credentials are compromised during the transfer.

***: The `BridgeManagementImpl` contract allows for direct ownership transfer through the `setOwner()` function. This approach can be risky as it does not provide a safeguard against accidental or malicious ownership changes. Implementing a two-step process for ownership transfer can enhance security by requiring the new owner to accept the ownership explicitly.
***: The contract `BridgeManagementImpl` does not implement a two-step process for transferring ownership. So ownership of the contract can be easily lost when making a mistake when transferring ownership.
***: It is possible that the `onlyOwner` role mistakenly transfers ownership to the wrong address, resulting in the loss of the `onlyOwner` role.
***:

Take a look at https://github.com/Secure3Audit/code_NeoX_Bridge_Contract/blob/f3764f6770d85e9438997630adac73f5314afbf4/code/contracts/management/BridgeManagementStorage.sol#L43-L46

```
    function _setOwner(address _owner) internal {
        owner = _owner;
    }
```

This function eventually gets called if the owner is to be swapped, issue however is that this implementation does this in one step, i.e there is no first settage of this admin and then the new admin claims the adminship by calling an acceptAdmin() function, whereas this can be considered a reckless current admin mistake, the risk to reward is extremely high, considering a loss of this adminship, means all the functionalities below would be inaccessible: *showcasing how protocol, becomes broken*
https://github.com/Secure3Audit/code_NeoX_Bridge_Contract/blob/f3764f6770d85e9438997630adac73f5314afbf4/code/contracts/management/BridgeManagementImpl.sol#L65-L128

```
function setOwner(address _owner) external onlyOwner {
        _setOwner(_owner);
        emit OwnerChange(_owner);
    }


    function setRelayer(address _relayer) external onlyOwner {
        _setRelayer(_relayer);
        emit RelayerChange(_relayer);
    }

    function setValidators(
        address[] calldata _validators,
        uint threshold
    ) external onlyOwner {
        _setValidators(_validators, threshold);
        emit ValidatorsChange(_validators, threshold);
    }


    function setGovernor(address _governor) external onlyOwner {
        _setGovernor(_governor);
        emit GovernorChange(_governor);
    }

    function setSecurityGuard(address _securityGuard) external onlyOwner {
        _setSecurityGuard(_securityGuard);
        emit SecurityGuardChange(_securityGuard);
    }

    function setFunder(address _funder) external onlyOwner {
        _setFunder(_funder);
        emit FunderChange(_funder);
    }
```

***: `_setOwner()` lets owner give the role to another address. This function has no checks and happens in a single step, which is prone to admin mistake.

## Recommendation

***:
Implement a two-step ownership transfer mechanism using OpenZeppelin's `Ownable2Step` to ensure secure and controlled ownership changes. This requires the new owner to explicitly accept ownership, preventing accidental or unauthorized transfers.

***:** Implement a two-step process for ownership transfer. This involves setting a pending owner and requiring the pending owner to accept the ownership explicitly.
***:** Consider using Ownable2StepUpgradeable(https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/access/Ownable2StepUpgradeable.sol).
***:** Recommend implementing a two-step process where the onwer nominates an account and the nominated account needs to call an `acceptOnwership()` function for the transfer of the ownership to fully succeed.
***:** Consider changing the admin in two steps as is commonly done in the web3 space, so break the setAdmin() function to two, 1. setAdmin(), 2. acceptAdmin(), where the new admin accepts their adminship.
***:** Use 2-step ownership transfer.

## Client Response

client response : Fixed.
This issue has been fixed in commit f59082feec130766905f3a64d4aeee4aed3777a9.
client response : Fixed. This issue has been fixed in commit f59082feec130766905f3a64d4aeee4aed3777a9.
client response : Fixed. This issue has been fixed in commit f59082feec130766905f3a64d4aeee4aed3777a9.

# NEO-18:No not `0` check

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Informational | Fixed | *** |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L201-L214

```
201: function setGasWithdrawalFee(uint256 _fee) external onlyGovernor {
202:         _setGasWithdrawalFee(_fee);
203:         emit GasWithdrawalFeeChange(_fee);
204:     }
205:
206:     function setMinGasWithdrawalAmount(uint256 _amount) external onlyGovernor {
207:         _setGasWithdrawalMinAmount(_amount);
208:         emit MinGasWithdrawalChange(_amount);
209:     }
210:
211:     function setMaxGasWithdrawalAmount(uint256 _amount) external onlyGovernor {
212:         _setGasWithdrawalMaxAmount(_amount);
213:         emit MaxGasWithdrawalChange(_amount);
214:     }
```

- code/contracts/bridge/BridgeStorage.sol#L256-L283
- code/contracts/bridge/BridgeStorage.sol#L278

```
256: function _setTokenWithdrawalFee(address _neoXToken, uint256 _fee) internal {
257:         tokenBridges[_neoXToken].config.fee = _fee;
258:     }
259:
260:     function _setTokenMinWithdrawalAmount(
261:         address _neoXToken,
262:         uint256 _amount
263:     ) internal {
264:         if (_amount > tokenBridges[_neoXToken].config.maxAmount)
265:             revert InvalidAmount();
266:         tokenBridges[_neoXToken].config.minAmount = _amount;
267:     }
268:
269:     function _setTokenMaxWithdrawalAmount(
270:         address _neoXToken,
271:         uint256 _amount
272:     ) internal {
273:         if (_amount < tokenBridges[_neoXToken].config.minAmount)
274:             revert InvalidAmount();
275:         tokenBridges[_neoXToken].config.maxAmount = _amount;
```

```
276:     }
277:
278:     function _setMaxTokenDeposits(
279:         address _neoXToken,
280:         uint256 _maxDeposits
281:     ) internal {
282:         tokenBridges[_neoXToken].config.maxDeposits = _maxDeposits;
283:     }
```

```
278: function _setMaxTokenDeposits(
```

- code/contracts/bridge/BridgeStorageV1.sol#L36-L49

```
36: constructor(address _management) {
37:         management = IBridgeManagement(_management);
38:         gasBridge = StorageTypes.GasBridge({
39:             paused: false,
40:             depositState: StorageTypes.State({nonce: 0, root: 0x0}),
41:             withdrawalState: StorageTypes.State({nonce: 0, root: 0x0}),
42:             config: StorageTypes.GasConfig({
43:                 fee: 1e17,
44:                 minAmount: 1e18,
45:                 maxAmount: 1e22,
46:                 maxDeposits: 100
47:             })
48:         });
49:     }
```

- code/contracts/library/TokenBridgeLib.sol#L79-L92

```
79: /**
80:      * @dev Validates the token configuration.
81:      * @param _config The token configuration.
82:      */
83:     function _isValidConfig(
84:         StorageTypes.TokenConfig memory _config
85:     ) internal pure returns (bool) {
86:         // The fee must always be greater than 0.
87:         return
88:             _config.fee > 0 &&
89:             _config.minAmount > 0 &&
90:             _config.maxAmount > _config.minAmount &&
91:             _config.maxDeposits > 0;
92:     }
```

## Description

***: The BridgeStorage contract manages various bridge functionalities, including gas and token bridges. The _setMaxGasDeposits function includes a zero check for the _maxDeposits parameter to ensure it is greater than zero. However, the _setMaxTokenDeposits function lacks this check, potentially allowing a zero value to be set, which could lead to unexpected behavior or errors.
***: Inside the `constructor` of the **BridgeStorageV1** contract, `gasBridge.config` is set:

```
            config: StorageTypes.GasConfig({
                fee: 1e17,
                minAmount: 1e18,
                maxAmount: 1e22,
                maxDeposits: 100
            })
```

These values can be set with the functions `setGasWithdrawalFee`, `setMinGasWithdrawalAmount` and `setMaxGasWithdrawalAmount` but these functions do not perform the check not `0` check.

In the case of the `fee` set to `0` with the function `withdrawGas` an attacker can do a DoS attack since `msg.value = 0` pass in the value check `if ((msg.value % 1e10) != 0) revert InvalidAmount();`, sending a large number of transactions.
In the case of `minAmount` and `maxAmount` break the functionality of the bridge.
***: When a token is registered in the `registerToken` function, it is checked for a valid configuration with the `TokenBridgeLib._isValidConfig` function:

```
    function _isValidConfig(
        StorageTypes.TokenConfig memory _config
    ) internal pure returns (bool) {
        // The fee must always be greater than 0.
        return
            _config.fee > 0 &&
            _config.minAmount > 0 &&
            _config.maxAmount > _config.minAmount &&
            _config.maxDeposits > 0;
    }
```

As we can see it is checked that these parameters are not `0`.
Then these values can be set with the functions `setTokenWithdrawalFee`, `setMinTokenWithdrawalAmount`, `setMaxTokenWithdrawalAmount` and `setMaxTokenDeposits` but these functions do not perform the check not `0` check.
In the case of the `fee` set to `0` with the function `withdrawToken` an attacker can do a DoS attack since the `msg.value` and `_amount` could be `0`, sending a large number of transactions.
In the case of `minAmount`, `maxAmount` and `maxDeposits` break the functionality of the bridge.

## Recommendation

***: To ensure consistency and prevent invalid configurations, a zero check should be added to the _setMaxTokenDeposits function. The updated function should look like this:

```
function _setMaxTokenDeposits(address _neoXToken, uint256 _maxDeposits) internal {
    if (_maxDeposits == 0) revert InvalidAmount();
    tokenBridges[_neoXToken].config.maxDeposits = _maxDeposits;
}
```

Adding a zero check in the _setMaxTokenDeposits function aligns it with the _setMaxGasDeposits function, ensuring consistent validation and preventing potential issues related to setting zero as the maximum deposits.
***:

```
@@ -185,17 +185,20 @@ contract BridgeStorage is BridgeStorageV1, UUPSUpgradeable {
     }

     function _setGasWithdrawalFee(uint256 _fee) internal {
+        if (_fee == 0) revert InvalidFee();
         if ((_fee % 1e10) != 0) revert InvalidFee();
         gasBridge.config.fee = _fee;
     }

     function _setGasWithdrawalMinAmount(uint256 _amount) internal {
+        if (_amount == 0) revert InvalidAmount();
         if ((_amount % 1e10) != 0) revert InvalidAmount();
         if (_amount >= gasBridge.config.maxAmount) revert InvalidAmount();
         gasBridge.config.minAmount = _amount;
     }

     function _setGasWithdrawalMaxAmount(uint256 _amount) internal {
+        if (_amount == 0) revert InvalidAmount();
         if ((_amount % 1e10) != 0) revert InvalidAmount();
         if (_amount <= gasBridge.config.minAmount) revert InvalidAmount();
         gasBridge.config.maxAmount = _amount;
```

***:

```
@@ -43,6 +43,7 @@ contract BridgeStorage is BridgeStorageV1, UUPSUpgradeable {
     error TokenBridgeNotRegistered(address neoXToken);
     error TokenWithdrawalFailed();
     error TransferFailed();
+    error InvalidMaxDeposits();

     // Modifiers for Role Restriction

@@ -254,6 +255,8 @@ contract BridgeStorage is BridgeStorageV1, UUPSUpgradeable {
     }

     function _setTokenWithdrawalFee(address _neoXToken, uint256 _fee) internal {
+        if (_fee == 0)
+            revert InvalidFee();
         tokenBridges[_neoXToken].config.fee = _fee;
     }

@@ -261,6 +264,8 @@ contract BridgeStorage is BridgeStorageV1, UUPSUpgradeable {
         address _neoXToken,
         uint256 _amount
     ) internal {
+        if (_amount == 0)
+            revert InvalidAmount();
         if (_amount > tokenBridges[_neoXToken].config.maxAmount)
             revert InvalidAmount();
         tokenBridges[_neoXToken].config.minAmount = _amount;
@@ -270,6 +275,8 @@ contract BridgeStorage is BridgeStorageV1, UUPSUpgradeable {
         address _neoXToken,
         uint256 _amount
     ) internal {
+        if (_amount == 0)
+            revert InvalidAmount();
         if (_amount < tokenBridges[_neoXToken].config.minAmount)
             revert InvalidAmount();
         tokenBridges[_neoXToken].config.maxAmount = _amount;
@@ -279,6 +286,8 @@ contract BridgeStorage is BridgeStorageV1, UUPSUpgradeable {
         address _neoXToken,
         uint256 _maxDeposits
     ) internal {
+        if (_maxDeposits == 0)
+            revert InvalidMaxDeposits();
         tokenBridges[_neoXToken].config.maxDeposits = _maxDeposits;
     }
```

## Client Response

client response : Fixed. This has been addressed in commit ca872b1eaac9bc153c57fd1001e4ee2d02eb4770.

client response : Fixed.
This issue is valid and will be fixed regarding the fee part.
A zero check is redundant for the maximum value, while a zero value is allowed as minimum. There might be cases where a zero value transfer could be needed since there are cases on Neo N3 where this is used, e.g., for claiming rewards for holding Neo. As long as the fee is not zero, this issue can be regarded as fixed.
This has been fixed in commit 357d22fddfdf36956617835a126890308939d406.

# NEO-19:Missing data length check in `_executeERC20Transfer`, potential silent failures can occur

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Informational | *** | |

## Code Reference

- code/contracts/library/TokenBridgeLib.sol#L15-L23

```
15: function _executeERC20Transfer(
16:         address _neoXToken,
17:         uint256 _amount,
18:         address _to
19:     ) internal returns (bool) {
20:         bytes memory transferCall = abi.encodeCall(IERC20.transfer, (_to, _amount));
21:         (bool success, bytes memory returndata) = address(_neoXToken).call(transferCall);
22:         return success && (returndata.length == 0 || abi.decode(returndata, (bool)));
23:     }
```

## Description

***: The `_executeERC20Transfer` function in the `BridgeImpl` contract lacks a check for `data.length >= 32` before decoding return data as a bool. This can lead to issues when interacting with non-standard ERC20 tokens that might return less than 32 bytes of data.

## Impact

- Attempting to decode short return data ( `< 32 bytes` ) as a bool could cause errors, leading to silent failed transfers.
- Misinterpreting short non-zero data as a successful bool return can result in incorrect assumptions about transfer success
- This function ( `_executeERC20Transfer` ) is used in `depositToken` and `claimToken` so it can directly impact the functionality of these functions. Silent failures can occur when dealing with non-standard ERC20 tokens.

## Recommendation

***: Add the `data.length >= 32` check to ensure robust handling of return data:

```
return success && (returndata.length == 0 || (returndata.length >= 32 && abi.decode(returndata, (bool))));
```

## Client Response

client response :

# NEO-20:Missing Zero Address Check

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Informational | Fixed | *** |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L443-L506

```
443: function withdrawToken(
444:         address _neoXToken,
445:         address _to,
446:         uint256 _amount
447:     )
448:         external
449:         payable
450:         override
451:         nonReentrant
452:         onlyBridgeUnpaused
453:         onlyTokenBridgeUnpaused(_neoXToken)
454:     {
455:         if (!_isRegisteredToken(_neoXToken))
456:             revert TokenBridgeNotRegistered(_neoXToken);
457:         StorageTypes.TokenConfig memory config = _getTokenConfig(_neoXToken);
458:         uint256 tokenValue = _amount;
459:         if (tokenValue < config.minAmount) revert InvalidAmount();
460:         if (tokenValue > config.maxAmount) revert InvalidAmount();
461:         if (msg.value < config.fee)
462:             revert InsufficientFee(msg.value, config.fee);
```

```
463:         _addUnclaimedRewards(msg.value);
464:
465:         // Execute the transfer of the tokens from the sender to the bridge contract.
466:         bool success = IERC20(_neoXToken).transferFrom(
467:             msg.sender,
468:             address(this),
469:             _amount
470:         );
471:         if (!success) revert TransferFailed();
472:
473:         // Compute the new root and update the token withdrawal state.
474:         StorageTypes.State memory state = _getTokenWithdrawalState(_neoXToken);
475:         uint256 newNonce = state.nonce + 1;
476:
477:         if (config.executionType == StorageTypes.ExecutionType.NEO) {
478:             if (tokenValue % 1e18 != 0) {
479:                 revert InvalidAmount();
480:             }
481:             tokenValue /= 1e18;
482:         }
```

```
483:
484:         bytes32 withdrawalHash = TokenBridgeLib._hashTokenBridgeOp(
485:             config.neoN3Token,
486:             _neoXToken,
487:             newNonce,
488:             _to,
489:             tokenValue
490:         );
491:         bytes32 newRoot = BridgeLib._computeNewRoot(state.root, withdrawalHash);
492:         _setTokenWithdrawalState(
493:             _neoXToken,
494:             StorageTypes.State({nonce: newNonce, root: newRoot})
495:         );
496:         emit TokenWithdrawal(
497:             _neoXToken,
498:             config.neoN3Token,
499:             newNonce,
500:             _to,
501:             tokenValue,
502:             msg.sender,
```

```
503:             withdrawalHash,
504:             newRoot
505:         );
506:     }
```

- code/contracts/bridge/BridgeStorageV1.sol#L36-L49

```
36: constructor(address _management) {
37:         management = IBridgeManagement(_management);
38:         gasBridge = StorageTypes.GasBridge({
39:             paused: false,
40:             depositState: StorageTypes.State({nonce: 0, root: 0x0}),
41:             withdrawalState: StorageTypes.State({nonce: 0, root: 0x0}),
42:             config: StorageTypes.GasConfig({
43:                 fee: 1e17,
44:                 minAmount: 1e18,
45:                 maxAmount: 1e22,
46:                 maxDeposits: 100
47:             })
48:         });
49:     }
```

- code/contracts/management/BridgeManagementStorage.sol#L16-L32

```
16: constructor(
17:         address _owner,
18:         address _relayer,
19:         uint256 _validatorThreshold,
20:         address[] memory _validators,
21:         address _governor,
22:         address _securityGuard,
23:         address _funder
24:     ) {
25:         _disableInitializers();
26:         _setOwner(_owner);
27:         _setRelayer(_relayer);
28:         _setValidators(_validators, _validatorThreshold);
29:         _setGovernor(_governor);
30:         _setSecurityGuard(_securityGuard);
31:         _setFunder(_funder);
32:     }
```

## Description

***: ## Summary
In the `withdrawToken` function, the `_to` parameter specifies the address to which the tokens should be sent on Neo N3. However, this address is not validated within the function. If `_to` is an invalid address, such as the zero address (0x0), the tokens will be sent to this address and effectively lost forever.

## Proof of concept

- A user approves and withdraws tokens to an address they mistakenly set as `0x0`.
- The function processes the withdrawal without validating the `_to` address.
- The tokens are transferred to 0x0 and lost permanently.

***: In Neo bridge contract and management contract constructor，they all forget to check address is zero:
BridgeStorageV1.sol:

```
    constructor(address _management) {
        management = IBridgeManagement(_management);
        gasBridge = StorageTypes.GasBridge({
            paused: false,
            depositState: StorageTypes.State({nonce: 0, root: 0x0}),
            withdrawalState: StorageTypes.State({nonce: 0, root: 0x0}),
            config: StorageTypes.GasConfig({
                fee: 1e17,
                minAmount: 1e18,
                maxAmount: 1e22,
                maxDeposits: 100
            })
        });
    }
```

BridgeManagementStorage.sol:

```
    constructor(
        address _owner,
        address _relayer,
        uint256 _validatorThreshold,
        address[] memory _validators,
        address _governor,
        address _securityGuard,
        address _funder
    ) {
        _disableInitializers();
        _setOwner(_owner);
        _setRelayer(_relayer);
        _setValidators(_validators, _validatorThreshold);
        _setGovernor(_governor);
        _setSecurityGuard(_securityGuard);
        _setFunder(_funder);
    }
```

In particular, in Contract BridgeManagementStorage, it has partially checked (validator):

```
    function _setValidators(
        address[] memory _validators,
        uint256 _threshold
    ) internal {
        uint256 validatorsLength = _validators.length;
        if (validatorsLength == 0) revert InvalidValidatorArray();
        if (_threshold == 0 || _threshold > validatorsLength)
            revert InvalidValidatorThreshold();
        for (uint256 i = 0; i < validatorsLength; i++) {
            if (_validators[i] == address(0)) revert InvalidAddress();
        }
        if (ManagementLib._hasDuplicates(_validators))
            revert InvalidValidatorArray();

        delete validators;
        for (uint256 i = 0; i < validatorsLength; i++) {
            validators.push(_validators[i]);
        }
        validatorThreshold = _threshold;
    }
```

but others is not:

```
    function _setRelayer(address _relayer) internal {
        relayer = _relayer;
    }

    function _setGovernor(address _governor) internal {
        governor = _governor;
    }

    function _setSecurityGuard(address _securityGuard) internal {
        securityGuard = _securityGuard;
    }

    function _setFunder(address _funder) internal {
        funder = _funder;
    }
```

so I think it's a finding.

## Recommendation

***: Add a check to ensure `_to` is a valid, non-zero address before proceeding with the token transfer.

```
if (_to == address(0)) revert InvalidAddress();
```

***: add address check in code，like this：

```
    constructor(address _management) {
        require(management != address(0));
        management = IBridgeManagement(_management);
        gasBridge = StorageTypes.GasBridge({
            paused: false,
            depositState: StorageTypes.State({nonce: 0, root: 0x0}),
            withdrawalState: StorageTypes.State({nonce: 0, root: 0x0}),
            config: StorageTypes.GasConfig({
                fee: 1e17,
                minAmount: 1e18,
                maxAmount: 1e22,
                maxDeposits: 100
            })
        });
    }
```

## Client Response

client response : Fixed. This has been fixed in commit da1dbf3811ee0511621dc1519e37d276aa074d15.

# NEO-21:Lack of check whether the token is registered

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Informational | Fixed | *** |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L274-L347

```
274: /**
275:      * @notice Distributes the provided Token that has been locked on Neo N3.
276:      * @dev The depositToken function is used to distribute tokens that have been locked on Neo N3.
277:      *      The deposits data need to be provided ordered based on their nonces.
278:      *      Before the deposits are distributed, the following steps are executed:
279:      *      - Check if the provided deposits are subsequent to the current nonce in storage and each othe
r.
280:      *      - Check if the computed root based on the provided deposits matches the provided root.
281:      *      - Check if the provided signatures are valid given the provided root and the current validato
rs.
282:      *      Once these checks are passed, the storage state is updated with the new nonce and root, and t
he deposits are distributed.
283:      * @param _neoXToken the address of the token on the Neo X network.
284:      * @param _tokenDepositRoot the new deposit root. The root must be the root that resulted from the la
st provided deposit.
285:      * @param _signatures the signatures of the validators. The signatures need to be ordered based on th
e order they have been stored in storage.
286:      * @param _deposits the deposit data. Each deposit's nonce, recipient address and the amount.
287:      */
288:     function depositToken(
289:         address _neoXToken,
290:         bytes32 _tokenDepositRoot,
291:         BridgeLib.Signature[] calldata _signatures,
292:         BridgeLib.DepositData[] calldata _deposits
293:     )
```

```
294:         external
295:         override
296:         onlyRelayer
297:         onlyBridgeUnpaused
298:         onlyTokenBridgeUnpaused(_neoXToken)
299:         nonReentrant
300:     {
301:         StorageTypes.State memory depositState = _getTokenDepositState(
302:             _neoXToken
303:         );
304:         StorageTypes.TokenConfig memory config = _getTokenConfig(_neoXToken);
305:
306:         // Check parameter validity
307:         uint depositLength = _deposits.length;
308:         if (depositLength == 0) revert InvalidDepositsLength();
309:         if (depositLength > config.maxDeposits) revert InvalidDepositsLength();
310:         // Check if provided deposit data's nonces are subsequent to the current nonce and each other.
311:         if (!BridgeLib._subsequentNonces(_deposits, depositState.nonce))
312:             revert InvalidNonceSequence();
313:         // Validate that the provided token deposit root is equal to the new computed root based on the p
rovided deposits.
```

```
314:         if (
315:             TokenBridgeLib._computeNewTopRoot(
316:                 depositState.root,
317:                 _getNeoN3Token(_neoXToken),
318:                 _neoXToken,
319:                 _deposits
320:             ) != _tokenDepositRoot
321:         ) revert InvalidRoot();
322:         // Verify that the provided signatures are valid given the provided deposit root and the current
validators.
323:         if (
324:             !management.verifyValidatorSignatures(
325:                 _tokenDepositRoot,
326:                 _signatures
327:             )
328:         ) revert InvalidValidatorSignatures();
329:
330:         // Update the token's deposit state
331:         _setTokenDepositState(
332:             _neoXToken,
333:             StorageTypes.State({
```

```
334:                 nonce: _deposits[depositLength - 1].nonce,
335:                 root: _tokenDepositRoot
336:             })
337:         );
338:         emit TokenDepositRootUpdate(
339:             _neoXToken,
340:             config.neoN3Token,
341:             _deposits[depositLength - 1].nonce,
342:             _tokenDepositRoot
343:         );
344:
345:         // Execute the token distribution
346:         _executeTokenDistribution(_neoXToken, config.executionType, _deposits);
347:     }
```

## Description

***: The function `depositToken` does not check whether the `_neoXToken` is registered before processing the deposit. If the `_neoXToken` is not registered, the `config.maxDeposits` will be 0 and the code will revert with `InvalidDepositsLength` error:

```
if (depositLength > config.maxDeposits) revert InvalidDepositsLength();
```

This error can be a nuisance to the user, leading them to believe that the `_deposits` is incorrect, while the real reason is that the `_neoXToken` is unregistered.

## Recommendation

***: Consider following fix:

```
function depositToken(
        address _neoXToken,
        bytes32 _tokenDepositRoot,
        BridgeLib.Signature[] calldata _signatures,
        BridgeLib.DepositData[] calldata _deposits
    )
        external
        override
        onlyRelayer
        onlyBridgeUnpaused
        onlyTokenBridgeUnpaused(_neoXToken)
        nonReentrant
    {
     if (!_isRegisteredToken(_neoXToken))
            revert TokenBridgeNotRegistered(_neoXToken);
       ...
       ...
```

## Client Response

client response : Fixed. This has been fixed in commit 264dbd005792e91294b7a1f3fee6ac0badc29dec.

# NEO-22:Consider allowing users to pass in a recipient address when claiming their gas

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Informational | Acknowledged | *** |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L145-L160
- code/contracts/bridge/BridgeImpl.sol#L388-L420

```
145: function claimGas(
146:         uint256 _nonce
147:     ) external onlyBridgeUnpaused onlyGasBridgeUnpaused nonReentrant {
148:         StorageTypes.Claimable memory claimable = _getGasClaimable(_nonce);
149:         uint256 amount = claimable.amount;
150:         address to = claimable.to;
151:         if (amount == 0) revert NonexistentClaimable();
152:         if (to == address(0)) revert NonexistentClaimable();
153:
154:         _deleteGasClaimable(_nonce);
155:         uint256 sendValue = GasBridgeLib._addTenDecimals(amount);
156:         (bool success, ) = to.call{value: sendValue}("");
157:         if (!success) revert TransferFailed();
158:         emit GasClaim(_nonce, to, amount);
159:     }
```

```
388: function claimToken(
389:         address _neoXToken,
390:         uint256 _nonce
391:     )
392:         external
393:         override
394:         onlyBridgeUnpaused
395:         onlyTokenBridgeUnpaused(_neoXToken)
396:         nonReentrant
397:     {
398:         StorageTypes.Claimable memory claimable = _getTokenClaimable(
399:             _neoXToken,
400:             _nonce
401:         );
402:         // Check if the claimable exists.
403:         address to = claimable.to;
404:         if (to == address(0)) revert NonexistentClaimable();
405:         _deleteTokenClaimable(_neoXToken, _nonce);
406:         StorageTypes.ExecutionType executionType = _getExecutionType(
407:             _neoXToken
```

```
408:         );
409:         assert(
410:             executionType == StorageTypes.ExecutionType.NEO ||
411:                 executionType == StorageTypes.ExecutionType.ERC20
412:         );
413:         // Note: For NEO tokens, the transfer value has already been extended with 18 decimals in the dep
osit function.
414:         bool success = TokenBridgeLib._executeERC20Transfer(
415:             _neoXToken,
416:             claimable.amount,
417:             to
418:         );
419:         if (!success) revert TransferFailed();
420:     }
```

## Description

***:

Take a look at https://github.com/Secure3Audit/code_NeoX_Bridge_Contract/blob/f3764f6770d85e9438997630adac73f5314afbf4/code/contracts/bridge/BridgeImpl.sol#L388-L420

```
    function claimToken(
        address _neoXToken,
        uint256 _nonce
    )
        external
        override
        onlyBridgeUnpaused
        onlyTokenBridgeUnpaused(_neoXToken)
        nonReentrant
    {
        StorageTypes.Claimable memory claimable = _getTokenClaimable(
            _neoXToken,
            _nonce
        );
        // Check if the claimable exists.
        //@audit
        address to = claimable.to;
        if (to == address(0)) revert NonexistentClaimable();
        _deleteTokenClaimable(_neoXToken, _nonce);
        StorageTypes.ExecutionType executionType = _getExecutionType(
            _neoXToken
        );
        assert(
            executionType == StorageTypes.ExecutionType.NEO ||
                executionType == StorageTypes.ExecutionType.ERC20
        );
        // Note: For NEO tokens, the transfer value has already been extended with 18 decimals in the deposi
t function.
        bool success = TokenBridgeLib._executeERC20Transfer(
            _neoXToken,
            claimable.amount,
            to
        );
        if (!success) revert TransferFailed();
    }
```

This function is used to claim tokens for users, issue however as hinted by the @audit tag is the fact that in this case the `to` address has been hardcoded, which means that users funds that are to be *claimed* could get stuck in protocol and lead to loss of funds/ *stuck funds* .

**Step-by-step POC:**

1. Tokens gets deposited for a user on NeoX .
2. Some time to passes.
3. User gets blacklisted by the token, this is possible with USDC/USDT.
4. User attempts to claim.
5. The execution would always revert on this line due to the user being blacklisted by Tether/Circle.

Note that from here: https://github.com/d-xo/weird-erc20#tokens-with-blocklists we can see that `USDC/USDT` employs a blocklisting feature.

***:
Take a look at https://github.com/Secure3Audit/code_NeoX_Bridge_Contract/blob/f3764f6770d85e9438997630adac73f5314afbf4/code/contracts/bridge/BridgeImpl.sol#L145-L160

```
    function claimGas(
        uint256 _nonce
    ) external onlyBridgeUnpaused onlyGasBridgeUnpaused nonReentrant {
        StorageTypes.Claimable memory claimable = _getGasClaimable(_nonce);
        uint256 amount = claimable.amount;
        address to = claimable.to;
        if (amount == 0) revert NonexistentClaimable();
        if (to == address(0)) revert NonexistentClaimable();

        _deleteGasClaimable(_nonce);
        uint256 sendValue = GasBridgeLib._addTenDecimals(amount);
        (bool success, ) = to.call{value: sendValue}("");
        if (!success) revert TransferFailed();
        emit GasClaim(_nonce, to, amount);
    }
```

This function is used in order to claim gas that has been deposited, issue however is that it's been hardcoded that the recipient must be the attached address during the deposit, this logic is generally frowned upon in the smart contract development space,.
In this instance however it could even be worse, considering a user who gets hacked, can never come claim his gas, cause the moment he does so, his account would get drained by the attackers.
I assume this should be QA-low level considering it's not an issue with the smart contracts themselves, however, having a recipient address ensures at all times the user can decide who/where he wants to send his tokens to, which in this case would mean he gets to save his tokens.

## Recommendation

***: Implement a mechanism that allows users to specify an alternate recipient address during the claiming process.
***: Consider attaching a recipient, but if you are to do so, a recipient should only be accepted in the case the caller of the function is the address where the deposits were attached to in the first place.

## Client Response

client response : Acknowledged.
This issue will be acknowledged.
We will fix this issue at a later stage.

# NEO-23:Better way to do the double for loop in BridgeManagementImpl.verifyValidatorSignatures()

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Code Style | Informational | Acknowledged | *** |

## Code Reference

- code/contracts/management/BridgeManagementImpl.sol#L53-L61

```
53: for (uint i = 0; i < threshold; i++) {
54:             for (j = n; j < validatorsLength; j++) {
55:                 if (recovered[i] == validators[j]) {
56:                     covered++;
57:                     break;
58:                 }
59:             }
60:             n = j + 1;
61:         }
```

## Description

***: This double for loop in verifyValidatorSignatures() does not follow common programming pattern:

```
        for (uint i = 0; i < threshold; i++) {
            for (j = n; j < validatorsLength; j++) {
                if (recovered[i] == validators[j]) {
                    covered++;
                    break;
                }
            }
            n = j + 1;
        }
```

The code looks hacky because it finds matching pairs within two arrays, and there isn't easy way to do this in Solidity. But if `validators` is a mapping, this code can be simplified into a single for loop.

## Recommendation

***: Consider using a mapping for `validators`, so double loop won't be needed.

## Client Response

client response : Acknowledged. We might fix this later.

# NEO-24:Attacker can abuse deposit nonce due to uint8 downcasting in BridgeLib._subsequentNonces(), users can lose fund

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Language Specific | Informational | Fixed | *** |

## Code Reference

- code/contracts/library/BridgeLib.sol#L23

```
23: for (uint8 i = 1; i <= depositsLength; i++) {
```

## Description

***: ### Impact
Users can lose fund if `_deposits` array is longer than 256 when `depositGas()` or `depositToken()` is called.

### Bug detail

The nonce check `_subsequentNonces()` uses uint8 for index in the for loop, which will be bounded by 256 entries:

```
    function _subsequentNonces(
        DepositData[] calldata _deposits,
        uint256 _startNonce
    ) internal pure returns (bool) {
        uint depositsLength = _deposits.length;
        // @audit-issue uint8 can overflow when _deposits array is longer than 256
        for (uint8 i = 1; i <= depositsLength; i++) {
            if (_deposits[i - 1].nonce != _startNonce + i) {
                return false;
            }
        }
        return true;
    }
```

Attacker can invoke a call with `_deposits` array longer than 256 entries to bypass nonce check, since indices beyond 255 are wrapped around to 0 due to modular arithmetic.
This is dangerous because

```
_addClaimableGas(depositEntry.nonce, to, depositEntry.amount);
```

will not behave as expected. This can be seen from the implementation of `_addClaimableGas()`:

```
function _addClaimableGas(
    uint256 _nonce,
    address _to,
    uint256 _amount
) internal {
    claimableGas[_nonce] = StorageTypes.Claimable({
        to: _to,
        amount: _amount
    });
}
```

If two entries in `_deposits` array have the same nonce (due to modular arithmetic), `claimableGas[_nonce]` will be overwritten by the second entry, therefore the info in the first entry is lost. In other words, the user whose deposit data is stored in the first entry cannot claim token and that asset is lost permanently.
Moreover, even from gas optimization perspective, uint8 isn't cheapter than uint256. Therefore there is no good reason to use uint8 here.

## Recommendation

***: Change the code to:

```
function _subsequentNonces(
    DepositData[] calldata _deposits,
    uint256 _startNonce
) internal pure returns (bool) {
    uint depositsLength = _deposits.length;
    - for (uint8 i = 1; i <= depositsLength; i++) {
    + for (uint256 i = 1; i <= depositsLength; i++) {
        if (_deposits[i - 1].nonce != _startNonce + i) {
            return false;
        }
    }
    return true;
}
```

## Client Response

client response : Fixed. This issue has been fixed in commit 0f7208130815e51a22bda46d30b5ef73ca7804b7.

# NEO-25:Accumulated variable `unclaimedRewards` that are not actually used

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Informational | Acknowledged | *** |

## Code Reference

- code/contracts/bridge/BridgeImpl.sol#L166-L176
- code/contracts/bridge/BridgeImpl.sol#L176
- code/contracts/bridge/BridgeImpl.sol#L443-L463
- code/contracts/bridge/BridgeImpl.sol#L463

```
166: function withdrawGas(
167:         address _to
168:     ) external payable onlyBridgeUnpaused onlyGasBridgeUnpaused {
169:         if (_to == address(0)) revert InvalidAddress();
170:         if ((msg.value % 1e10) != 0) revert InvalidAmount();
171:         StorageTypes.GasConfig memory config = _getGasBridgeConfig();
172:         StorageTypes.State memory state = _getGasBridgeWithdrawalState();
173:         uint256 actualWithdrawalAmount = msg.value - config.fee;
174:         if (actualWithdrawalAmount < config.minAmount) revert InvalidAmount();
175:         if (actualWithdrawalAmount > config.maxAmount) revert InvalidAmount();
176:         _addUnclaimedRewards(config.fee);
```

```
176: _addUnclaimedRewards(config.fee);
```

```
443: function withdrawToken(
444:         address _neoXToken,
445:         address _to,
446:         uint256 _amount
447:     )
448:         external
449:         payable
450:         override
451:         nonReentrant
452:         onlyBridgeUnpaused
453:         onlyTokenBridgeUnpaused(_neoXToken)
454:     {
455:         if (!_isRegisteredToken(_neoXToken))
456:             revert TokenBridgeNotRegistered(_neoXToken);
457:         StorageTypes.TokenConfig memory config = _getTokenConfig(_neoXToken);
458:         uint256 tokenValue = _amount;
459:         if (tokenValue < config.minAmount) revert InvalidAmount();
460:         if (tokenValue > config.maxAmount) revert InvalidAmount();
461:         if (msg.value < config.fee)
462:             revert InsufficientFee(msg.value, config.fee);
```

```
463:         _addUnclaimedRewards(msg.value);
```

```
463: _addUnclaimedRewards(msg.value);
```

- code/contracts/bridge/BridgeStorage.sol#L116-L118

```
116: function _addUnclaimedRewards(uint256 _amount) internal {
117:         unclaimedRewards += _amount;
118:     }
```

## Description

***: The functions `withdrawGas` and `withdrawToken` will charge a fee:

```
_addUnclaimedRewards(msg.value);
```

```
_addUnclaimedRewards(config.fee);
```

The function `_addUnclaimedRewards` just adds the fee to `unclaimedRewards`:

```
function _addUnclaimedRewards(uint256 _amount) internal {
        unclaimedRewards += _amount;
    }
```

There is no function to withdraw these fees. That means these fees will be locked in the contract.
***: The functions `withdrawGas` and `withdrawToken` of the **BridgeImpl** contract receive `GAS` (native) to pay the gas fee which is registered in the variable `unclaimedRewards`.
The problem is that there is no way to withdraw these rewards.
***: The bridge contract collects transaction fees through the _addUnclaimedRewards() function, but does not provide an interface for authorized addresses to withdraw these accumulated fees.
Whenever a withdrawal transaction is executed, the fee amount is deducted and added to the `unclaimedRewards` balance through `_addUnclaimedRewards()`. However, there is no corresponding function implemented for authorized addresses like the governor or funder to retrieve these fees.

## Recommendation

***: Consider adding a function to withdraw fees.
***:

```
@@ -26,6 +26,7 @@ interface IGasBridge {
     event MinGasWithdrawalChange(uint256 newAmount);
     event MaxGasWithdrawalChange(uint256 amount);
     event MaxGasDepositsChange(uint8 amount);
+    event WithdrawRewards(address to, uint256 amount);

     function pauseGasBridge() external;
```

```
@@ -218,6 +218,16 @@ contract BridgeImpl is BridgeStorage, IBridge, IGasBridge, ITokenBridge {
         emit MaxGasDepositsChange(_maxNrDeposits);
     }

+    function withdrawRewards(address _to) external onlyGovernor {
+        uint256 amount = unclaimedRewards;
+
+        delete unclaimedRewards;
+
+        (bool success, ) = _to.call{value: amount}("");
+        if (!success) revert TransferFailed();
+        emit WithdrawRewards(_to, amount);
+    }
+
     // ITokenBridge Implementation
```

***: Implement a `withdrawFees()` function that allows authorized addresses to withdraw a specified amount from unclaimedRewards, similar to other management functions. Proper validation such as available balance check should be included.

## Client Response

client response : Acknowledged.
At the current stage, it has not been decided what exactly will be done with the collected fees, e.g., how exactly they will be distributed to participating entities of the bridge operation. In order to keep the amount of collected fees transparent and avoid some off-chain computation at a later stage, it was decided that this value is used to keep track already now in a transparent setting. Since the bridge contract is upgradeable, a withdrawFee function can be added in a future version of the implementation contract.

# 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 Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.