Nume Security review

Pashov Audit Group

Conducted by: Dan Ogurtsov, T1M0H, Todorov

November 28th 2023 - December 10th 2023

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Introduction 3
4. About nume-p2p-contracts 3
5. Risk Classification 4
   5.1. Impact 4
   5.2. Likelihood 4
   5.3. Action required for severity levels 4
6. Security Assessment Summary 5
7. Executive Summary 6
8. Findings 9
   8.1. Critical Findings 9
      [C-01] Attack to force the project into the exodusMode 9
      [C-02] Attacker can turn on Exodus Mode 10
   8.2. High Findings 12
      [H-01] User can repeat calling withdrawExodus() to steal all tokens 12
      [H-02] NFT Withdrawal request is not deleted after successful withdrawal 13
      [H-03] Breaking the deposit limit with different msg.value and _user 15
      [H-04] Stuck tokens during the Exodus Mode 16
   8.3. Medium Findings 18
      [M-01] WITHDRAWAL_REQUEST_TIMEOUT cannot be too short 18
      [M-02] The same signature is used for both withdrawal request and subscription cancelling 18
      [M-03] Setting deposit limit to zero stops all new deposits 20
      [M-04] Withdrawal_Stake and pending deposits are returned to wrong address 21
      [M-05] Staked WITHDRAWAL_STAKE cannot be withdrawn during the exodusMode 22
      [M-06] Unexpected gains&losses for pending withdrawals if WITHDRAWAL_STAKE is changed 23
   8.4. Low Findings 24
      [L-01] getUserDepositsLeft() and getUserNftDepositsLeft() can revert after setting new limit 24

[L-02] currBlockNumber naming - is not related to block.number 24
[L-03] totalTokens doesn't count native coin 25
[L-04] nftDepositsLimit can be bypassed with malicious ERC721 25
[L-05] Two contractOwner roles, likely with not intended behavior and risk of mistakes 26
[L-06] It will become difficult to use signatures when Nume block time decreases, because signature is valid within 1 Nume block 27
[L-07] setDepositsLimit - if decreased, may have ongoing userDepositCount above the new limit 28
[L-08] RegistrationFacet.changeVerificationAddress() zero check missed 28
[L-09] enforceExodusMode() naming 29
[L-10] PaymentUtils.pay() misses safeTransfer() 29
[L-11] depositsQueueHash includes nullified invalid deposits 29
[L-12] receive() msg.value is hard to return 29

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **nume-p2p-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About nume-p2p-contracts

The Nume Protocol is a Layer-3 on Polygon zkEVM supporting token transfers and NFT mints/trades. The contracts look for the state update to be attested by a specific predefined public key, which is determined by the validators of the Nume Protocol and corresponds to a private key known only by the Trusted Execution Environment running the off-chain state update algorithm. Smart contracts in the scope are built using the Diamond pattern on zkEVM and allow depositing NFTs and supported ERC20 tokens, submitting withdrawal requests, and handling these pending requests.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

***review commit hash* - 50d6bd5a29bc0fcce8852baadd5f63603670dcee**

***fixes review commit hash* -**

Scope

The following smart contracts were in scope of the audit:

- `NumeP2P`
- `libraries/LibDiamond`
- `libraries/AppStorage`
- `utils/NumeNFTFactory`
- `utils/MerkleUtils`
- `utils/PaymentUtils`
- `utils/VerifyUserUtils`
- `utils/QueueUtils`
- `utils/ECDSAUtils`
- `utils/MerkleUtils`
- `facets/nume/NFTDepositsFacet`
- `facets/nume/DepositsFacet`
- `facets/nume/WithdrawalsFacet`
- `facets/nume/RegistrationFacet`
- `facets/nume/ConfigFacet`
- `facets/nume/SettlementsFacet`
- `facets/diamond/OwnershipFacet`

7. Executive Summary

Over the course of the security review, Dan Ogurtsov, T1M0H, Todorov engaged with Nume to review nume-p2p-contracts. In this period of time a total of **24** issues were uncovered.

Protocol Summary

Protocol Name	nume-p2p-contracts
Repository	https://github.com/nume-crypto/nume-p2p-contracts
Date	November 28th 2023 - December 10th 2023
Protocol Type	payments protocol

Findings Count

Severity	**Amount**
Critical	2
High	4
Medium	6
Low	12
Total Findings	**24**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Attack to force the project into the exodusMode	Critical	Resolved
[C-02]	Attacker can turn on Exodus Mode	Critical	Resolved
[H-01]	User can repeat calling withdrawExodus() to steal all tokens	High	Resolved
[H-02]	NFT Withdrawal request is not deleted after successful withdrawal	High	Resolved
[H-03]	Breaking the deposit limit with different msg.value and _user	High	Resolved
[H-04]	Stuck tokens during the Exodus Mode	High	Resolved
[M-01]	WITHDRAWAL_REQUEST_TIMEOUT cannot be too short	Medium	Resolved
[M-02]	The same signature is used for both withdrawal request and subscription cancelling	Medium	Resolved
[M-03]	Setting deposit limit to zero stops all new deposits	Medium	Acknowledged
[M-04]	Withdrawal_Stake and pending deposits are returned to wrong address	Medium	Resolved
[M-05]	Staked WITHDRAWAL_STAKE cannot be withdrawn during the exodusMode	Medium	Resolved
[M-06]	Unexpected gains&losses for pending withdrawals if WITHDRAWAL_STAKE is changed	Medium	Resolved
[L-01]	getUserDepositsLeft() and getUserNftDepositsLeft() can revert after setting new limit	Low	Resolved
[L-02]	currBlockNumber naming - is not related to block.number	Low	Resolved

[L-03]	totalTokens doesn't count native coin	Low	Resolved
[L-04]	nftDepositsLimit can be bypassed with malicious ERC721	Low	Resolved
[L-05]	Two contractOwner roles, likely with not intended behavior and risk of mistakes	Low	Resolved
[L-06]	It will become difficult to use signatures when Nume block time decreases, because signature is valid within 1 Nume block	Low	Resolved
[L-07]	setDepositsLimit - if decreased, may have ongoing userDepositCount above the new limit	Low	Resolved
[L-08]	RegistrationFacet.changeVerificationAddress() zero check missed	Low	Resolved
[L-09]	enforceExodusMode() naming	Low	Resolved
[L-10]	PaymentUtils.pay() misses safeTransfer()	Low	Resolved
[L-11]	depositsQueueHash includes nullified invalid deposits	Low	Acknowledged
[L-12]	receive() msg.value is hard to return	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Attack to force the project into the exodusMode

Severity

Impact: High, disabled core project functions forever

Likelihood: High, easy to execute by anyone

Description

The attack flow:

1. Deposit ETH from a malicious smart contract A, which reverts when receiving ETH
2. Await for the deposit settlement to Nume network
3. call `submitWithdrawalRequest()`
4. the owner has 14 days to withdraw via `notarizeSettlement()`
5. the owner tries to settle the withdrawal but it fails - a malicious smart contract A intentionally reverts not receiving ETH
6. The attacker waits 14 days
7. The attacker calls `challengeUserWithdrawal()` => `ns.isInExodusMode = true;`

As a result, the check `enforceExodusMode` will not pass in core Nume functions - enabling a Mass exit scenario for the project, which means no new deposits and force withdrawals enabled (with no withdrawal requests on Nume). Also, there is no way to set `isInExodusMode` back to false. (Actually, it exists - urgently develop a new facet with the new code, deploy, and run the fix) The same issue can happen with ERC20 tokens with blacklists, for example `USDC`.

Another attacker vector with the same principle - avoid the deposit being invalidated. If an ETH deposit by a user is tagged as invalid (during `notarizeSettlement`), Nume pays back the deposit. If such a malicious deposit is among valid deposits - there will be no way to remove it, thus it will be a problem to process valid deposits. So the owner will have to treat such a deposit as valid.

Recommendations

It is better to let users withdraw their ETH by themselves, in a separate function. Instead of transferring funds directly, the function can increment a mapping like `user=>token=>amount`. Then, users have to call the function to withdraw this "balance". Also, consider introducing some instruments to disable `exoduceMode`.

[C-02] Attacker can turn on Exodus Mode

Severity

Impact: High. Protocol stops operating.

Likelihood: High. Nothing stops from performing this attack.

Description

Attacker can reenter Nume when `Witdrawal_Stake` is returned on settlement notarization, and in `fallback()` call `challengeUserWithdrawal()` to activate Exodus Mode.

`notarizeSettlement()` processes withdrawals submitted on-chain, such on-chain submissions require `Withdrawal_stake` which is returned on successful withdrawal. Note that it firstly deletes timestamp of that `withdrawalRequest`, but updates `ns.lastFinalizedWithdrawalsQueueIndex` in the very end after all withdrawals:

```
for (uint256 i; i < numContractWithdrawals; ) {
                ...
                delete ns.withdrawalRequests[
                    _args.contractWithdrawalAddresses[i]
                ][_args.contractWithdrawalTokenAddresses[i]];
                delete ns.withdrawalRequestTimestamps[
@>                  ns.lastFinalizedWithdrawalsQueueIndex + i + 1
                ];
                if (_args.contractWithdrawalAmounts[i] > 0) {
                    PaymentUtils.pay(
                        _args.contractWithdrawalAddresses[i],
                        _args.contractWithdrawalTokenAddresses[i],
                        _args.contractWithdrawalAmounts[i]
                    );
                    PaymentUtils.pay(
                        _args.contractWithdrawalAddresses[i],
                        0x1111111111111111111111111111111111111111,
                        ns.WITHDRAWAL_STAKE
                    );
                }
                ...
            }
            ...
@>          ns.lastFinalizedWithdrawalsQueueIndex = _args
                .handledWithdrawalsQueueIndex;
```

It introduces attack vector:

1. Attacker creates Malicious contract, which in `fallback()` calls `WithdrawalsFacet.challengeUserWithdrawal()`

2. Attacker on-chain submits withdrawal to that malicious contract
3. Operator calls `notarizeSettlement()` and processes withdrawals
4. `Withdrawal_Stake` is transferred to that Malicious contracts and function `challengeUserWithdrawal()` is called

Now let's have a look on `challengeUserWithdrawal()`. 1) As you remember, `ns.lastFinalizedWithdrawalsQueueIndex` is updated in the very end, therefore current `queueIndex` is not finalized. 2) Timestamp was cleared before transferring assets. As a result, all requires are passed and protocol enters Exodus Mode

```
function challengeUserWithdrawal(
        uint256 _queueIndex,
        bool _isNft
    ) external {
        AppStorage.NumeStorage storage ns = AppStorage.numeStorage();

        if (_isNft) {
            ... // Skipped because scenario describes ERC20 withdrawal
        } else {
            require(
                _queueIndex <= ns.currWithdrawalsQueueIndex &&
@>                  _queueIndex > ns.lastFinalizedWithdrawalsQueueIndex,
                "WithdrawalsFacet: Invalid queue index"
            );
            require(
                block.timestamp >
@>                  ns.withdrawalRequestTimestamps[_queueIndex] +
                        ns.WITHDRAWAL_REQUEST_TIMEOUT,
                "WithdrawalsFacet: Withdrawal request has not expired yet"
            );
        }

        ns.isInExodusMode = true;
        emit ExodusModeEntered(_queueIndex, _isNft);
    }
```

Recommendations

Add `nonReentrant` modifier on all external functions of Nume.

8.2. High Findings

[H-01] User can repeat calling `withdrawExodus()` to steal all tokens

Severity

Impact: High. Any user can steal all funds.

Likelihood: Medium. Exodus Mode must be enabled.

Description

Function `withdrawExodus()` is used to withdraw funds deposited to Nume when protocol enters Exodus Mode, i.e. when operator doesn't perform bridging from Nume to Polygon.

This function only verifies that user had token balance prior to the last settlement. If yes - transfers requested tokens to user. However nothing stops user from replaying call to `withdrawExodus()`.

```
function withdrawExodus(
        WithdrawalRequestArgs calldata _args
    ) external nonReentrant {
        AppStorage.NumeStorage storage ns = AppStorage.numeStorage();
        if (!ns.isInExodusMode) {
            revert AppStorage.NotInExodusMode();
        }
        require(
            ECDSAUtils.recoverSigner(
                abi.encodePacked(_args.user, ns.currBlockNumber),
                _args.signature
            ) == _args.user,
            "WithdrawExodus: Invalid user signature"
        );

        VerifyUserUtils.verifyUser(...);

        if (_args.isNft) {
            PaymentUtils.payNft(
                _args.user,
                _args.tokenAddress,
                _args.balanceOrTokenId,
                _args.mintedNft
            );
        } else {
            require(
                _args.balanceOrTokenId > 0,
                "WithdrawalsFacet: Invalid balance"
            );
            PaymentUtils.pay(
                _args.user,
                _args.tokenAddress,
                _args.balanceOrTokenId
            );
        }
        ...
    }
```

Suppose following scenario:

1. User deposited 300 USDC
2. Nume enters Exodus Mode
3. Now user can perform infinite withdraws of his 300 USDC

Recommendations

Keep track of withdrawn amounts in `withdrawExodus()`. For example introduce internal `mapping (address token => uint256 withdrawnAmount)`

[H-02] NFT Withdrawal request is not deleted after successful withdrawal

Severity

Impact: High. Attacker can steal user's nfts.

Likelihood: Medium. It requires re-deposit of NFT

Description

Withdrawal backend nft request is incorrectly deleted after performing withdrawal.

Here you can see that delete is performed on memory array instead of storage:

```
function withdrawNFT(
        address _user,
        address _nftContractAddress,
        uint256 _tokenId,
        bool _mintedNft,
        uint256 _queueIndex,
        bool _isContractWithdrawal
    ) external nonReentrant {
        AppStorage.enforceExodusMode();

        AppStorage.NumeStorage storage ns = AppStorage.numeStorage();

        bytes32 queueItem = keccak256(
            abi.encodePacked(_user, _nftContractAddress, _tokenId, _mintedNft)
        );

        if (_isContractWithdrawal) {
            ...
        } else {
@>          bytes[] memory userBackendNftWithdrawalRequests = ns
                .userBackendNftWithdrawalRequests[_user];
            require(
                _queueIndex <= userBackendNftWithdrawalRequests.length,
                "NFTWithdrawalsFacet: Invalid queue index"
            );
            (
                address user,
                address nftContractAddress,
                uint256 tokenId,
                bool mintedNft
            ) = abi.decode(
                    userBackendNftWithdrawalRequests[_queueIndex - 1],
                    (address, address, uint256, bool)
                );
            PaymentUtils.payNft(user, nftContractAddress, tokenId, mintedNft);
@>          delete userBackendNftWithdrawalRequests[_queueIndex - 1];
        }

        emit NFTWithdrawn(_user, _nftContractAddress, _tokenId, _mintedNft);
    }
```

It means actually user's withdrawal request is not deleted after withdrawal. It allows user to withdraw the same nft multiple times. Suppose following scenario:

1. User1 deposits NFT.
2. User1 withdraws NFT via `withdrawNFT()`. His request is still.
3. User1 sells NFT to User2.
4. User2 deposits NFT. But now User1 has ability withdraw it.

Recommendations

```
function withdrawNFT(
        address _user,
        address _nftContractAddress,
        uint256 _tokenId,
        bool _mintedNft,
        uint256 _queueIndex,
        bool _isContractWithdrawal
    ) external nonReentrant {
        ...

        if (_isContractWithdrawal) {
            ...
        } else {
            bytes[] memory userBackendNftWithdrawalRequests = ns
                .userBackendNftWithdrawalRequests[_user];
            require(
                _queueIndex <= userBackendNftWithdrawalRequests.length,
                "NFTWithdrawalsFacet: Invalid queue index"
            );
            ...
            PaymentUtils.payNft(user, nftContractAddress, tokenId, mintedNft);
-           delete userBackendNftWithdrawalRequests[_queueIndex - 1];
+           delete ns.userBackendNftWithdrawalRequests[_user][_queueIndex - 1];
        }

        emit NFTWithdrawn(_user, _nftContractAddress, _tokenId, _mintedNft);
    }
```

[H-03] Breaking the deposit limit with different msg.value and _user

Severity

Impact: Medium, a daily limit bypassed (invariant broken)

Likelihood: High, easily available behavior

Description

Nume tries to limit the number of deposits per day for a user in two contracts - `NFTDepositsFacet` and `DepositsFacet`. During the deposit, we have this code (slightly rewritten for simplicity):

```
...
uint256 currentTime = block.timestamp;
if (currentTime - ns.userDepositTimestamp[_user] >= 1 days) {
            delete ns.userDepositCount[_user];
            ns.userDepositTimestamp[_user] = currentTime;
        }
require ( ns.userDepositCount[msg.sender] < ns.depositsLimit )
...
ns.userDepositCount[msg.sender]++;
_deposit(_user, 0x1111111111111111111111111111111111111111, msg.value);
```

As you can see, the function deletes the counter for the `_user` daily:

```
delete ns.userDepositCount[_user];
```

But later, the function checks the counter for the `msg.sender` and does `++` for `msg.sender`:

```
require ( ns.userDepositCount[msg.sender] < ns.depositsLimit )
...
ns.userDepositCount[msg.sender]++;
```

`userDepositCount[_user]` is never checked to be below `ns.depositsLimit`, and is never incremented. As a result, `ns.depositsLimit` can easily be bypassed by changing msg.senders.

Moreover, ns.userDepositCount[msg.sender] is always incremented, and never deleted/nullified, even daily.

Recommendations

Replace `ns.userDepositCount[msg.sender]` with `ns.userDepositCount[_user]` in both deposit functions - `depositERC20()` and `deposit()`. Make sure that `msg.sender` is only used as a token sender. This problem exists in both deposit contracts - `NFTDepositsFacet` and `DepositsFacet`. **OR** Maybe it was the intention to limit exactly `msg.sender`. To prevent spamming. If yes, `delete ns.userDepositCount[_user];` should be replaced with `delete ns.userDepositCount[msg.sender];`, because the counter for msg.sender is never nullified.

[H-04] Stuck tokens during the Exodus Mode

Severity

Impact: High, tokens stuck

Likelihood: Medium, it must be pending withdrawals, and the Exodus Mode enabled

Description

During the Exodus Mode users can prove their Nume balance and call `withdrawExodus()` to withdraw NFTs not yet withdrawn and that have not received approval in `notarizeSettlement()` yet. Also, users can have some funds already approved. In this case, they have to call `withdrawNFT()` manually, which is the final step to finalize the approved withdrawals. But this function has the following line:

```
AppStorage.enforceExodusMode();
```

It means that these approved withdrawals will not be finalized during the exodus mode:

- `withdrawNFT()` will fail, as not allowed during the exodus mode
- `withdrawExodus()` will not prove that users have these NFT

Recommendations

Remove `AppStorage.enforceExodusMode();` from `withdrawNFT()`

8.3. Medium Findings

[M-01] WITHDRAWAL_REQUEST_TIMEOUT cannot be too short

Severity

Impact: High, the protocol will stop

Likelihood: Low, the variable has a default value and unlikely to be revised

Description

`OwnershipFacet` has `setWithdrawalRequestTimeout()` which sets `WITHDRAWAL_REQUEST_TIMEOUT`.

`WITHDRAWAL_REQUEST_TIMEOUT` is a highly risky parameter. If it is too low, there will be more likely to fall into exodusMode - which means the protocol is disabled (no functions to get back from the exodusMode, the protocol will require deploying a new Diamond or adding new facets).

Recommendations

We recommend setting a minimum allowed value for `WITHDRAWAL_REQUEST_TIMEOUT` or disabling revisions from the default 14 days.

[M-02] The same signature is used for both withdrawal request and subscription cancelling

Severity

Impact: Medium. User can't specify what action to perform when gives signature. As a result, user can't use signatures in trustless manner because tx sender can perform different action on behalf of user.

Likelihood: Medium. Usage of signature is impacted, however user can send transaction on his own to avoid problems.

Description

Here you can see that signature in both methods contains the same parameters:

```
function submitWithdrawalRequest(
        WithdrawalRequestArgs calldata _args
    ) external payable nonReentrant {
        AppStorage.enforceExodusMode();
        AppStorage.NumeStorage storage ns = AppStorage.numeStorage();
        ...
        require(
            ECDSAUtils.recoverSigner(
@>              abi.encodePacked(_args.user, ns.currBlockNumber),
                _args.signature
            ) == _args.user,
            "SubmitWithdrawalRequest: Invalid user signature"
        );
       ...
    }

    function cancelSubscriptionRequest(
        CancelSubscriptionRequestArgs calldata _args
    ) external {
        AppStorage.enforceExodusMode();

        AppStorage.NumeStorage storage ns = AppStorage.numeStorage();

        require(
            ECDSAUtils.recoverSigner(
@>              abi.encodePacked(_args.user, ns.currBlockNumber),
                _args.signature
            ) == _args.user,
            "SettlementsFacet: Invalid user signature"
        );
        ...
    }
```

It means that transmitter who sends transaction, can withdraw user's funds though user signed to perform subscription cancelling and vice versa. Possibility of altering data breaks possibility of trustless use of signatures.

Recommendations

Add unique salt to signature arguments, for example method name:

```
function submitWithdrawalRequest(
        WithdrawalRequestArgs calldata _args
    ) external payable nonReentrant {
        AppStorage.enforceExodusMode();
        AppStorage.NumeStorage storage ns = AppStorage.numeStorage();
        ...
        require(
            ECDSAUtils.recoverSigner(
-               abi.encodePacked(_args.user, ns.currBlockNumber),
+               abi.encodePacked
+ (_args.user, ns.currBlockNumber, "submitWithdrawalRequest"),
                _args.signature
            ) == _args.user,
            "SubmitWithdrawalRequest: Invalid user signature"
        );
       ...
    }

    function cancelSubscriptionRequest(
        CancelSubscriptionRequestArgs calldata _args
    ) external {
        AppStorage.enforceExodusMode();

        AppStorage.NumeStorage storage ns = AppStorage.numeStorage();

        require(
            ECDSAUtils.recoverSigner(
-               abi.encodePacked(_args.user, ns.currBlockNumber),
+               abi.encodePacked
+ (_args.user, ns.currBlockNumber, "cancelSubscriptionRequest"),
                _args.signature
            ) == _args.user,
            "SettlementsFacet: Invalid user signature"
        );
        ...
    }

    function withdrawExodus(
        WithdrawalRequestArgs calldata _args
    ) external nonReentrant {
        AppStorage.NumeStorage storage ns = AppStorage.numeStorage();
        if (!ns.isInExodusMode) {
            revert AppStorage.NotInExodusMode();
        }
        require(
            ECDSAUtils.recoverSigner(
-               abi.encodePacked(_args.user, ns.currBlockNumber),
+               abi.encodePacked
+ (_args.user, ns.currBlockNumber, "withdrawExodus"),
                _args.signature
            ) == _args.user,
            "WithdrawExodus: Invalid user signature"
        );
        ...
}
```

[M-03] Setting deposit limit to zero stops all new deposits

Severity

Impact: High, new deposit will be stopped (DoS)

Likelihood: Low, zero input by mistake or with the intention to cancel/disable deposit limit

Description

Deposit limits are managed with functions `setDepositsLimit()` and `setNftDepositsLimit()`. They set `depositsLimit` and `nftDepositsLimit` without zero input checks. Zero can be inputted either by mistake, or with the intention to disable limits.

Recommendations

We recommend requiring that the new value is not zero.

[M-04] `Withdrawal_Stake` and pending deposits are returned to wrong address

Severity

Impact: Low. Withdrawal initiator loses 0.01 ETH per withdrawal

Likelihood: High. The only prerequisite is to perform withdrawal on behalf of another user, which is expected behavior

Description

Currently `Withdrawal_Stake` is returned to user whose withdrawal is processed, instead of actual sender who submitted withdrawal request and staked it.

Here you can see that `Withdrawal_Stake` is returned to withdrawal receiver, instead of withdrawal initiator:

```
function notarizeSettlement(
        NotarizeSettlementArgs calldata _args
    ) external nonReentrant {
            ...
            for (uint256 i; i < numContractWithdrawals; ) {
                ...
                if (_args.contractWithdrawalAmounts[i] > 0) {
                    PaymentUtils.pay(
                        _args.contractWithdrawalAddresses[i],
                        _args.contractWithdrawalTokenAddresses[i],
                        _args.contractWithdrawalAmounts[i]
                    );
@>                  PaymentUtils.pay(
                        _args.contractWithdrawalAddresses[i],
                        0x1111111111111111111111111111111111111111,
                        ns.WITHDRAWAL_STAKE
                    );
                }
                ...
            }
            for (uint256 i; i < numContractWithdrawals; ) {
                ...
                if (_args.isContractNftWithdrawalValid[i]) {
@>                  PaymentUtils.pay(
                        _args.contractNftWithdrawalAddresses[i],
                        0x1111111111111111111111111111111111111111,
                        ns.WITHDRAWAL_STAKE
                    );
                } else {
                    ...
                }
                ...
            }
        ...
    }
```

Recommendations

Introduce parameter `sender` in withdrawals requests. Return `Withdrawal_Stake` to `sender` of request instead of withdrawal receiver.

[M-05] Staked WITHDRAWAL_STAKE cannot be withdrawn during the exodusMode

Severity

Impact: Medium, a portion of user funds lost

Likelihood: Medium, exodusMode is a scenario, but not so likely

Description

`submitWithdrawalRequest()` requires provided `msg.value`=`WITHDRAWAL_STAKE` in order to have a pending withdrawal. It is designed to be returned when a withdrawal request is approved via `notarizeSettlement()`. But during the mass exit scenario, there is no way to return staked `WITHDRAWAL_STAKE`. Only pending deposits and verified Nume balance can be withdrawn, but not `WITHDRAWAL_STAKE` if any.

Recommendations

[M-06] Unexpected gains&losses for pending withdrawals if WITHDRAWAL_STAKE is changed

Severity

Impact: High, potential funds stolen from other users and DoS

Likelihood: Low, as `Withdrawal_Stake` is not so likely to be changed, must be increased, and must have pending withdrawals

Description

`submitWithdrawalRequest()` receives `msg.value` as `WITHDRAWAL_STAKE`. However, it does not store the exact value received per request. When withdrawal requests are proceeded in `notarizeSettlement()`, the current `WITHDRAWAL_STAKE` is returned. Thus if `WITHDRAWAL_STAKE` was updated between "submit" and "notarize" (via `setWithdrawStake()`), the new updated value will be sent back, which is different from initially staked. As a result, pending withdrawals will experience either a loss or a gain. If `WITHDRAWAL_STAKE` decreases - users will receive less than staked (loss) If `WITHDRAWAL_STAKE` increases - users will receive more than staked (gain)

Gains for such users mean a loss for the whole contract - lack of funds to finalize all withdrawals in case of a mass exit scenario (DoS).

Some extravagant scenarios include the frontrun attack:

1. (Frontrun) Withdrawal request, stake a smaller value
2. `WITHDRAWAL_STAKE` increases
3. waiting for the request to proceed, receive the increased value

Recommendations

There are a few options:

1. store staked value per user or per request
2. make sure that there are no pending withdrawals when calling `setWithdrawStake()`
3. disable changing `WITHDRAWAL_STAKE`

8.4. Low Findings

[L-01] `getUserDepositsLeft()` and `getUserNftDepositsLeft()` can revert after setting new limit

Getters for user's deposits left can revert in `NFTDepositsFacet` and `DepositsFacet` if limit was set during current epoch, while user's deposit count is greater than new limit:

```
function setDepositsLimit(uint256 _limit) external {
        AppStorage.enforceIsOwner();
        AppStorage.numeStorage().depositsLimit = _limit;
        emit DepositLimit(_limit);
    }

    function getUserDepositsLeft(
        address _user
    ) external view returns (uint256) {
        AppStorage.NumeStorage storage ns = AppStorage.numeStorage();
        if (block.timestamp - ns.userDepositTimestamp[_user] >= 1 days) {
            return ns.depositsLimit;
        }
@>      return ns.depositsLimit - ns.userDepositCount[_user];
    }
```

```
function setNftDepositsLimit(uint256 _limit) external {
        AppStorage.enforceIsOwner();
        AppStorage.numeStorage().nftDepositsLimit = _limit;
        emit NFTDepositLimit(_limit);
    }

    function getUserNftDepositsLeft(
        address _user
    ) external view returns (uint256) {
        AppStorage.NumeStorage storage ns = AppStorage.numeStorage();
        if (block.timestamp - ns.userNftDepositTimestamp[_user] >= 1 days) {
            return ns.nftDepositsLimit;
        }
@>      return ns.nftDepositsLimit - ns.userNftDepositCount[_user];
    }
```

Recommendation: handle case when current limit is greater than user's count and return 0 instead.

[L-02] currBlockNumber naming - is not related to block.number

`currBlockNumber` is incremented by one every time `notarizeSettlement()` is called. It also has the getter getCurrBlockNumber() with the comment `Returns the current`

`block number.` But this variable is never compared to the actual `block.number`. The usecase of this variable is to be rather a nonce. Consider renaming the variable to have no relation to `block.number`.

[L-03] `totalTokens` doesn't count native coin

Consider including native coin too.

Test file can be found here

Recommendation:

```
contract NumeP2P {
    constructor(
        address _contractOwner,
        address _diamondCutFacet,
        bytes memory _data
    ) payable {
        ...

+       ++ns.totalTokens;
        ns.supportedTokens[0x1111111111111111111111111111111111111111] = true;
        ...
    }
...
}
```

[L-04] `nftDepositsLimit` can be bypassed with malicious ERC721

Here it doesn't follow CEI pattern and perform effects before external call. ERC721 can be malicious and reenter function `depositNFT()`, bypassing limit. Because `ns.userNftDepositCount` is updated in the very end.

```
function depositNFT(
        address _user,
        address _nftContractAddress,
        uint256 _tokenId
    ) external {
        AppStorage.enforceExodusMode();
        AppStorage.NumeStorage storage ns = AppStorage.numeStorage();

@>      if (!_checkNftDepositStatus(_user)) {
            revert AppStorage.ExceededMaximumDailyCalls(_user);
        }

@>      IERC721(_nftContractAddress).safeTransferFrom(
            msg.sender,
            address(this),
            _tokenId
        );
@>      ns.userNftDepositCount[msg.sender]++;
        _depositNFT(_user, _nftContractAddress, _tokenId);
    }
```

Recommendation: Make external call in the very end:

```
function depositNFT(
        address _user,
        address _nftContractAddress,
        uint256 _tokenId
    ) external {
        AppStorage.enforceExodusMode();
        AppStorage.NumeStorage storage ns = AppStorage.numeStorage();

        if (!_checkNftDepositStatus(_user)) {
            revert AppStorage.ExceededMaximumDailyCalls(_user);
        }

-       IERC721(_nftContractAddress).safeTransferFrom(
-           msg.sender,
-           address(this),
-           _tokenId
-       );
        ns.userNftDepositCount[msg.sender]++;
        _depositNFT(_user, _nftContractAddress, _tokenId);
+       IERC721(_nftContractAddress).safeTransferFrom(
+           msg.sender,
+           address(this),
+           _tokenId
+       );
    }
```

[L-05] Two contractOwner roles, likely with not intended behavior and risk of mistakes

Nume diamond has two `contractOwner` addresses, stored in two different storage, both are used in different situations:

- AppStorage.numeStorage().contractOwner
- LibDiamond.diamondStorage().contractOwner

owner() returns `LibDiamond.diamondStorage().contractOwner`

getContractOwner() public returns `AppStorage.numeStorage().contractOwner`

contractOwner() internal, LibDiamond returns `LibDiamond.diamondStorage().contractOwner`

transferOwnership() `msd.sender` should be `LibDiamond.diamondStorage().contractOwner` but sets a new address to `LibDiamond.diamondStorage().contractOwner`

setContractOwner() public `msd.sender` should be `LibDiamond.diamondStorage().contractOwner` sets a new address to `AppStorage.numeStorage().contractOwner`

setContractOwner() internal, LibDiamond sets a new address to `LibDiamond.diamondStorage().contractOwner`

enforceIsOwner(), internal, AppStorage checks `msd.sender` should be `AppStorage.numeStorage().contractOwner`

enforceIsContractOwner() internal, LibDiamond checks `msd.sender` should be `LibDiamond.diamondStorage().contractOwner`

Given the same naming, the probability of the mistake is sufficient.

Some suspicious example of mixed usage is:

`ConfigFacet.supportToken()` uses `LibDiamond.enforceIsContractOwner()` but `DepositFacet.setDepositsLimit()` uses `AppStorage.enforceIsOwner()`

The very similar namings make management confusing and increase the probability of mistakes.

Some examples of possible mistakes:

- using `setContractOwner()` instead of `transferOwnership`, when changing `LibDiamond.diamondStorage().contractOwner`
- confusing between `enforceIsOwner()` and `enforceIsContractOwner` when developing new facets
- external users experiencing different results when calling `owner()` and `getContractOwner()`

Consider having only one `owner` role. **OR** If now everything works as intended, and two different roles are required for operation, we strongly recommend renaming variables and functions. For example:

- `LibDiamond.diamondStorage().contractOwner` => "diamondOwner"
- `AppStorage.numeStorage().contractOwner` => "numeOwner"

[L-06] It will become difficult to use signatures when Nume block time decreases, because signature is valid within 1 Nume block

Signature is valid when used in the exact Nume block when it was signed. If Nume block time decreases due to increasing volume, it will mean decrease of valid signature time. Currently signature is valid only in current Nume block:

```
function withdrawExodus(
        WithdrawalRequestArgs calldata _args
    ) external nonReentrant {
        ...
        require(
            ECDSAUtils.recoverSigner(
@>              abi.encodePacked(_args.user, ns.currBlockNumber),
                _args.signature
            ) == _args.user,
            "WithdrawExodus: Invalid user signature"
        );
        ...
    }

    function submitWithdrawalRequest(
        WithdrawalRequestArgs calldata _args
    ) external payable nonReentrant {
        ...
        require(
            ECDSAUtils.recoverSigner(
@>              abi.encodePacked(_args.user, ns.currBlockNumber),
                _args.signature
            ) == _args.user,
            "SubmitWithdrawalRequest: Invalid user signature"
        );
        ...
    }
```

Recommendation: Consider specifying Nume block expiration as signature arguments, instead of using current block.

[L-07] setDepositsLimit - if decreased, may have ongoing userDepositCount above the new limit

`DepositsFacet.setDepositsLimit()` sets AppStorage.numeStorage().depositsLimit If this value decreased, some users with high enough `userDepositCount` can be left with `userDepositCount` potentially above the new `depositsLimit`. `getUserDepositsLeft()` will revert (underflow).

Same thing for `NFTDepositsFacet.setNftDepositsLimit()`

[L-08] RegistrationFacet.changeVerificationAddress() zero check missed

`RegistrationFacet.changeVerificationAddress()` allows setting `AppStorage.numeStorage().verificationAddress` as the zero address. It is not correct for future signature checks used in `SettlementsFacet`.

[L-09] enforceExodusMode() naming

`AppStorage.sol` functions have the rule for naming of `enforce`+`{Rule}`.

Correct examples:

- enforceIsOwner()
- enforceIsGovernor()

Wrong example:

- enforceExodusMode()

It is wrong because in fact the code checks that the contract is not in the exodusMode:

```
function enforceExodusMode() internal view {
        require(
            !numeStorage().isInExodusMode,
            "Nume: Protocol is in exodus mode"
        );
    }
```

So it is better to rename to e.g. `enforceNotExodusMode()`.

[L-10] PaymentUtils.pay() misses safeTransfer()

Contracts usually use `safeTransfer()` and `safeTransferFrom()`. `PaymentUtils.pay()` uses transfer directly. It is more secure to use `safeTransfer()` there.

[L-11] depositsQueueHash includes nullified invalid deposits

`notarizeSettlement()` in the beginning of the function deletes invalid deposits (their data is nullified). Then, valid deposits are settled in the sequential order. `computeQueueHash()` can include previously deleted deposits, thus `settlementMessage` will include some data as zero bytes.

Make sure if it is an expected behavior, and that from the side of the Nume these zero bytes are proceeded correctly. If not, one of the options is to skip empty value in `computeQueueHash()`.

[L-12] receive() msg.value is hard to return

`NumeP2P` has the default `receive()` function, which does not have any code. Received Native token there will not be accounted in the storage, and there are no functions to handle this received value.

This value will be hard to remove - either introducing new facets with new functions, or using `backendWithdrawalAddresses` (which is not intended behavior for this scenario).

Consider some of the options:

1. delete `receive()`
2. treating sending native token as a deposit
3. if additional native token on balance is required for some not revealed operations, introduce new functions to handle these operations