



BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

# FINAL REPORT

Stader Labs

MaticX

September 2024

## Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

## 1. Project Details

Important:

Please ensure that the deployed contract matches the source-code of the last commit hash.

| Project           | Stader - MaticX   |
|-------------------|---|
| Website           | staderlabs.com  |
| Language          | Solidity  |
| Methods           | Manual Analysis   |
| Github repository | <a href="https://github.com/stader-labs/maticX/tree/6889ca1b630e294131660c83816078d74465a7a0/contracts">https://github.com/stader-labs/maticX/tree/6889ca1b630e294131660c83816078d74465a7a0/contracts</a> |
| Resolution 1      |   |

## 2. Detection Overview

| Severity      | Found | Resolved | Partially Resolved | Acknowledged (no change made) |
|---------------|-------|----------|--------------------|-------------------------------|
| High          | 6     |          |                    |                               |
| Medium        | 3     |          |                    |                               |
| Low           | 5     |          |                    |                               |
| Informational | 9     |          |                    |                               |
| Governance    | 2     |          |                    |                               |
| Total         | 25    |          |                    |                               |

### 2.1 Detection Definitions

| Severity      | Description  |
|---------------|--|
| High          | The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users. |
| Medium        | While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.                                |
| Low           | Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately  |
| Informational | Effects are small and do not post an immediate danger to the project or users  |
| Governance    | Governance privileges which can directly result in a loss of funds or other potential undesired behavior   |

### 3. Detection

#### Global

|                       |   |  |
|-----------------------|---|--|
| Issue_01              | Reminder: Storage Layout correctness  |  |
| Severity              | Informational   |  |
| Description           | <p>Since both audited contracts are meant to be implementation contracts for proxies which upgrade previous iterations. It is mandatory to ensure that the proxy layout is not accidentally crashed by inheriting new dependencies / contracts.</p> <p>One can simply use <a href="#">hardhat-storage-layout</a> or <a href="#">forge layout storage CONTRACT</a></p> |  |
| Recommendations       | Consider keeping this in mind   |  |
| Comments / Resolution |   |  |

## ValidatorRegistry

The `ValidatorRegistry` contract is a registry contract that maintains a list of validator IDs for the `MaticX` Liquid Staking Architecture. It allows administrators (addresses with the `DEFAULT_ADMIN_ROLE`) to add and remove validators from the registry, ensuring that only active validators with a valid share contract are included. The contract keeps track of preferred validators for deposits and withdrawals, which can be set by accounts with the `BOT` role to determine delegations.

Before adding a validator, the contract verifies that the validator exists in the `StakeManager` and is active. When removing a validator, it ensures that the validator is not set as a preferred validator and that it has no remaining stake associated with the `MaticX` contract. The registry provides functions to retrieve the list of validators and specific validator IDs, facilitating interaction with other contracts in the staking ecosystem.

Additionally, the contract incorporates access control mechanisms using OpenZeppelin's `AccessControlUpgradeable`, allowing role-based permissions. It also includes pausability through `PausableUpgradeable`, enabling the contract to be paused and unpaused by administrators for maintenance or emergency situations which prevents adding/removing and setting preferred validators.

## Privileged Functions

- `grantRole (onlyRole)`
- `revokeRole (onlyRole)`
- `initializeV2`
- `addValidator`
- `removeValidator`
- `setPreferredDepositValidatorId`
- `setPreferredWithdrawalValidatorId`
- `setMaticX`
- `setVersion`
- `togglePause`

|                 |   |
|-----------------|---|
| Issue_02        | DoS of <b>removeValidator</b> by dusting a small amount of <b>ValidatorShare</b> tokens to the <b>MaticX</b> contract   |
| Severity        | Medium  |
| Description     | <p>The <b>removeValidator</b> function allows for removing any validator which has a zero-balance and is not a preferred validator:</p> <pre> require(     preferredDepositValidatorId != _validatorId,     "Can't remove a preferred validator for deposits" ); require(     preferredWithdrawalValidatorId != _validatorId,     "Can't remove a preferred validator for withdrawals" );  address validatorShare = stakeManager.getValidatorContract(     _validatorId ); (uint256 validatorBalance, ) = IValidatorShare(validatorShare) .getTotalStake(maticX); require(validatorBalance == 0, "Validator has some shares left"); </pre> <p>A malicious user can simply purchase a small amount of the <b>ValidatorShare</b> token, transfer them to <b>MaticX</b> which then results in a revert of the following check:</p> <pre> require(validatorBalance == 0, "Validator has some shares left"); </pre> <p>and essentially prevents the removal from validators.</p> |
| Recommendations | Consider implementing a <b>_ignoreBalance</b> boolean which optionally allows to bypass this check in such scenarios.   |

|                              |   |
|------------------------------|---|
|                              | <p>Optionally, it is also possible to execute a migration followed by a <code>removeValidator</code> call (in the same transaction) to remove any dusted validators or execute a temporary change of the <code>MaticX</code> address (which can also be dusted if not removed within the same transaction).</p> |
| <b>Comments / Resolution</b> |   |

|                              |  |
|------------------------------|--|
| <b>Issue_03</b>              | DoS of <code>removeValidator</code> by delegating outstanding rewards to the “to be removed” <code>_validatorId</code> (if BOT within <code>MaticX</code> is not trusted)  |
| <b>Severity</b>              | <b>Low</b>   |
| <b>Description</b>           | <p>In a similar mechanism as the “DoS of <code>removeValidator</code> by dusting a small amount of <code>ValidatorShare</code> tokens to the <code>MaticX</code> contract” issue, it is possible to dust validators by invoking the <code>stakeRewardsAndDistributeFees</code> function and buy shares from different validators, since this function lacks a check for the preferred depositor.</p> <p>This is however only possible if any address with the <b>BOT</b> role (<b>within <code>MaticX</code>; not to confuse with the <b>BOT</b> role within the <code>ValidatorRegistry</code></b>) is not trusted.</p> |
| <b>Recommendations</b>       | Consider implementing a <code>_ignoreBalance</code> boolean which optionally allows to bypass this check in such scenarios.  |
| <b>Comments / Resolution</b> |  |



## MaticX

The **MaticX** contract is the Liquid Staking solution which was developed by Stader for the Polygon Staking Architecture. Users can deposit **POL** or **MATIC** tokens in exchange for **MaticX** tokens, following the rule of three based on the overall staked **POL** token amount of **MaticX** and the circulating **MaticX** supply:

>  $\text{POLAmount} * \text{MaticXSupply} / \text{stakedPOL}$

>  $\text{MaticXAmount} * \text{stakedPol} / \text{MaticXSupply}$

The **MaticX** contract basically serves as a delegator which stakes these tokens through different validators based on the preferred setting within the **ValidatorRegistry** contract. These stakes will then earn a share of the validator rewards which can be either claimed via the **withdrawRewards** / **withdrawValidatorsReward** functions or automatically whenever a new deposit or withdrawal request is happening.

Once rewards have been claimed they will be staked in the same manner as the normal token deposit flow by invoking the **buyVoucherPOL** function which increases the underlying staked **POL** amount without minting any **MaticX** tokens and therefore it increases the value of **MaticX** by increasing the exchange rate.

A treasury fee which is by default 5% is taken on these rewards whenever the **stakeRewardsAndDistributeFees** function is invoked. This treasury fee is the revenue stream for Stader.

Users can redeem their **MaticX** tokens by invoking the **requestWithdraw** function which burns **MaticX** and creates an **unbond** request on the **ValidatorShare** contract which can be claimed after the **WITHDRAWAL\_DELAY** has been passed, by invoking **claimWithdrawal**.

## Appendix: StakeManager

The **StakeManager** contract is a core component of Polygon's staking architecture. It manages the registration and lifecycle of validators, allowing them to stake tokens, participate in consensus, and earn rewards.

Delegators can also stake their tokens through validators to earn a share of the rewards. The contract handles staking, unstaking, delegation, reward distribution, validator auctions, and updates to the validator set.

The part of interest for our auditing process is only the **staking through delegation mechanism**, where delegators stake tokens through validators. This is facilitated via the `updateValidatorState` function to increase or decrease the overall delegated amount towards a validator.

Delegator rewards are then distributed based on this amount and the corresponding owned `ValidatorShare` tokens, which is handled within `withdrawDelegatorsReward` and `delegatorsReward`.

Topping-up delegator rewards is handled within `_increaseValidatorRewardWithDelegation` which is connected to the `checkSignatures` function. Furthermore, the contract is epoch-based, starting by epoch 1 and the epoch is incremented whenever `checkSignatures` is invoked.

This contract is not included in the audit scope.

## Appendix: ValidatorShare

The `ValidatorShare` contract is the first instance where the `MaticX` contract is interacting with and is tied to a specific validator. When depositing `POL/MATIC` tokens into the Polygon staking architecture, the `ValidatorShare` ERC20 token is received which serves as staking receipt and can later be redeemed for `POL/MATIC` tokens. In the current iteration, the `ValidatorShare` token does not accrue any value, it has a steady `exchangeRate` and `withdrawRate` of 100 or  $1e29$ . Rewards can be directly claimed by the token owner and the token is freely transferable via the standard `transfer` function (but not via `transferFrom`). Furthermore, slashing is currently disabled.

`MATIC` and `POL` tokens are worth exactly the same and can be considered as (technically) the same token. The contract exposes binary call-paths to honor the interaction for both tokens which is primarily for backwards compatibility.

To facilitate deposits the **ValidatorShare** contract exposes the **buyVoucher** and **buyVoucherPol** functions.

To facilitate withdrawal requests, the **ValidatorShare** contract exposes the **sellVoucher\_newPOL** function

To facilitate request claims, the **ValidatorShare** contract exposes the **unstakeClaimTokens\_newPOL** function

To facilitate reward claiming, the **ValidatorShare** contract exposes the **withdrawRewardsPol** functions.

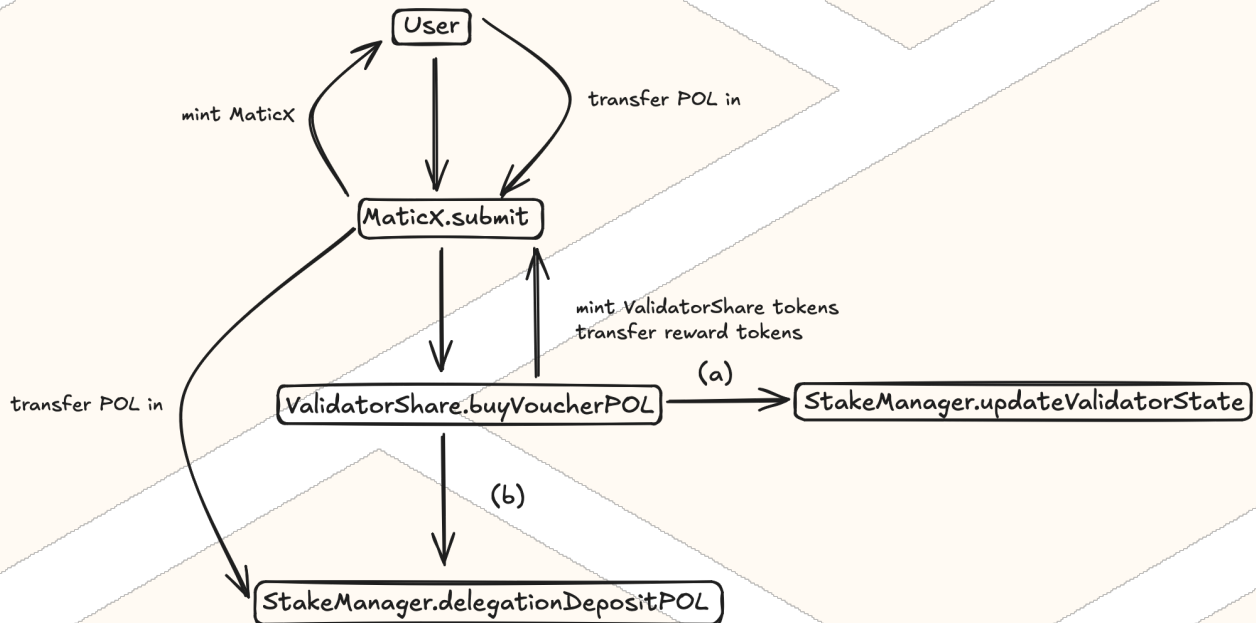
Furthermore this contract exposes several other functions but the above mentioned functions are the only ones used by the **MaticX** contract.

This contract is not included in the audit scope.

## Appendix: Deposit Flow

Users can deposit **POL/MATIC** tokens via the **submit** function which then mints the corresponding amount of **MaticX** to the user and delegates the stake to a validator. During delegation, the corresponding amount of **ValidatorShare** tokens is minted to the **MaticX** contract.

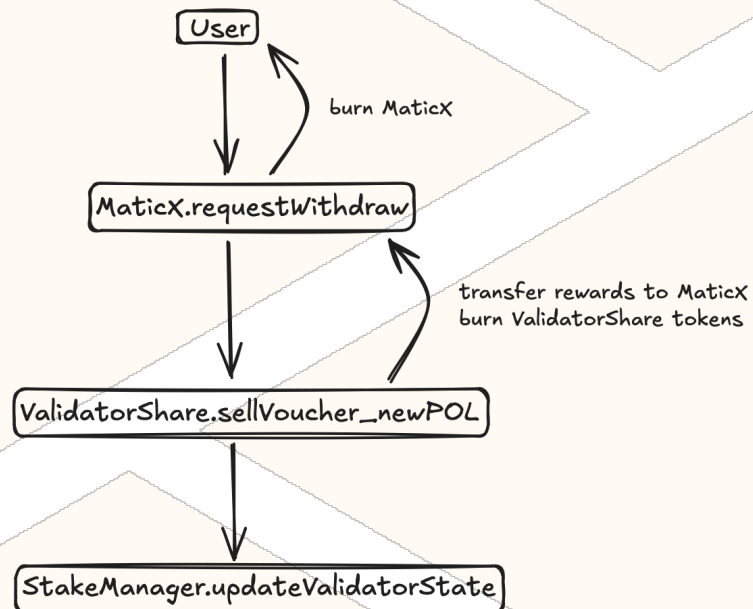
The flow is as follows:



## Appendix: Request Flow

Users can redeem **MaticX** tokens via the **requestWithdraw** function which then burns the provided amount of **MaticX** tokens and creates a withdrawal request on the **ValidatorShare** contract which is claimable by the user once the delay has surpassed. Since there can be a scenario where the balance of the preferred validator is insufficient to honor the withdrawal amount, a loop is executed which considers subsequent validators to honor the accurate withdrawal amount. This can therefore result in more than one withdrawal request being created. The **validatorNonce** within the **WithdrawalRequest** is corresponding to the **unbondNonce** within the **ValidatorShare** contract, ensuring that only the initial requester can claim the finalized withdrawal request.

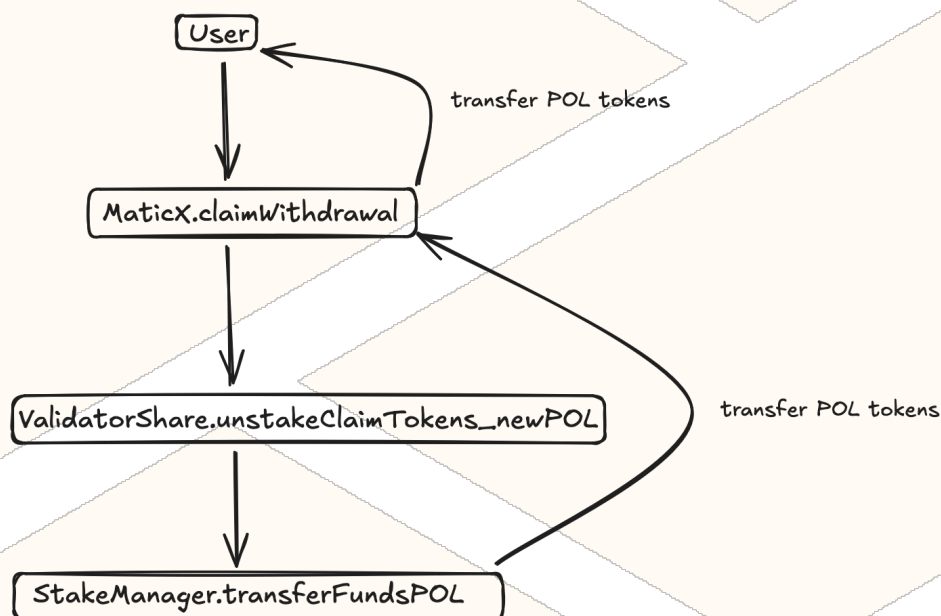
The flow is as follows:



## Appendix: Claim Flow

Once a request has been successfully created, the request creator can claim this request whenever the `requestEpoch` has surpassed by calling `claimWithdrawal`. This will then trigger the `unstakeClaimTokens_newPOL` function in the corresponding `ValidatorShare` contract which transfers the requested funds to the `MaticX` contract and then towards the caller.

The flow is as follows:



## Appendix: Binary POL/MATIC solution

The **MaticX** as well as the **ValidatorShare** contract allow for depositing **MATIC** as well as **POL**. Both tokens are handled exactly the same with the only difference that **MATIC** tokens will be migrated to **POL** tokens using a 1:1 ratio, whenever **MATIC** is staked. The call-paths are similar and binary.

## Appendix: Migrate Delegation

The **MaticX** contract allows for migrating delegated stakes from one validator to another validator. This is trivially done by calling the **migrateDelegation** function which then invokes the **migrateDelegation** function on the **StakeManager**. Funds are just migrated via simple share burns and mints.

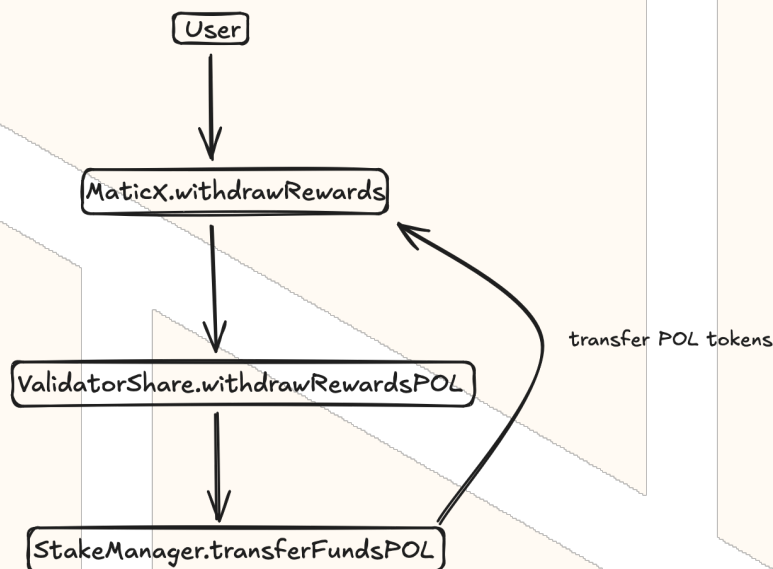
## Appendix: Reward Mechanism

The contract accrues rewards based on the delegated stake on each validator. The reward calculation is handled within each **ValidatorShare** contract but follows a simple masterchef-like pattern where rewards are distributed based on the overall supply distribution of **ValidatorShare** tokens.

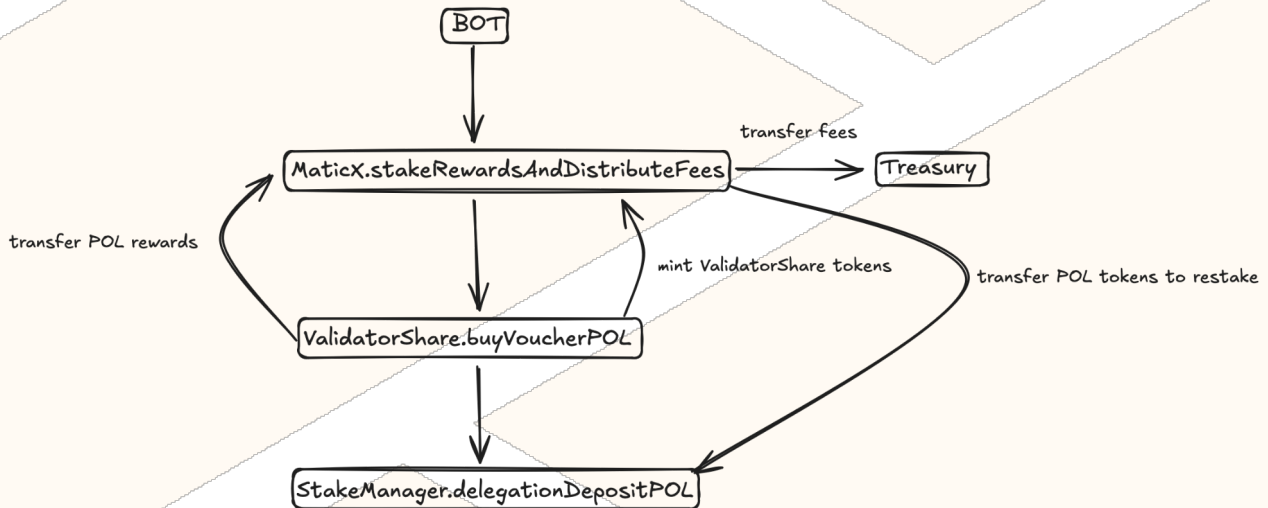
Rewards can be simply claimed by anyone via invoking **withdrawRewards** (for one validator) or **withdrawValidatorsReward**, which then claims rewards from the corresponding validator(s) via the **withdrawRewardsPOL** function and transfers these to the **MaticX** contract (this also automatically happens upon each **sellVoucher** / **buyVoucher** call and during the transfer of any **ValidatorShare** tokens).

Once rewards have been received, anyone with the **BOT** role can invoke the **stakeRewardsAndDistributeFees** function which takes a small fee to the treasury and delegates the leftover amount to the desired validator. The deposit will increase the exchange rate.

The flow for claiming rewards is as follows:



The flow for staking rewards is as follows:



## Appendix: Core Invariants

The core invariants of the protocol are the following:

- 1) Deposits should always increase the `totalMaticXSupply` by  

$$> \text{depositAmount} * \text{totalMaticXSupply} / \text{stakedPol}$$
- 2) Deposits should never influence the **MaticX** exchange rate
- 3) Withdrawals should always decrease `stakedPol` by  

$$> \text{maticXRedeemed} * \text{stakedPol} / \text{totalMaticXSupply}$$
- 4) Withdrawals should never influence the **MaticX** exchange rate
- 5) The exchange rate should always be up to date before any deposit/withdraw request
- 6) Request claims should always transfer out the exact same amount as requested



- 7) Requests should always match with the corresponding **unbondNonce** within the **ValidatorShare** contract
- 8) Reward compounds should always positively influence the **MaticX** exchange rate by increasing the underlying staked **POL** amount
- 9) Withdrawal requests should be claimable once the delay has been surpassed
- 10) Withdrawal requests can only be claimed once

## Privileged Functions

- grantRole
- revokeRole
- initializeV2
- migrateDelegation
- setFeePercent
- setTreasury
- setValidatorRegistry
- setFxStateRootTunnel
- setVersion
- togglePause

|                              |   |
|------------------------------|---|
| <b>Issue_04</b>              | Governance Privilege: Contract owner has control over funds   |
| <b>Severity</b>              | <b>Governance</b>   |
| <b>Description</b>           | <p>Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. This includes several functionalities such as pausing, changing the registry and more.</p> <p>Furthermore, this contract is used as proxy implementation which grants the proxy admin full control over all user funds</p> |
| <b>Recommendations</b>       | Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.  |
| <b>Comments / Resolution</b> |   |

|                       |  |
|-----------------------|--|
| Issue_05              | Governance of <b>ValidatorShare</b> and <b>StakeManager</b> contracts has several privileges that can negatively impact <b>MaticX</b>  |
| Severity              | Governance   |
| Description           | <p>The Polygon staking architecture has several privileges which grant governance full control over all funds within the contract. Some functionalities can prevent buy/sell whereas some functionalities can result in a loss of funds. We have aggregated these functions:</p> <p>For <b>ValidatorShare</b>:</p> <ul style="list-style-type: none"> <li>a) migrateOut</li> <li>b) migrateIn</li> <li>c) updateDelegation</li> </ul> <p>For <b>StakeManager</b>:</p> <ul style="list-style-type: none"> <li>a) setDelegationEnabled</li> <li>b) setStakingToken</li> <li>c) unstake/unstakePOL</li> <li>d) drain</li> </ul> <p>There are possibly also several other potential contract states where shares cannot be bought/sold. Overall it must be clear that in the worst case scenario, all funds can be lost.</p> |
| Recommendations       | We do not recommend a change. We assume that the Polygon team is highly trusted.   |
| Comments / Resolution |  |

|             |   |
|-------------|---|
| Issue_06    | Lack of reward compounding before submit allows users to extract value from the protocol  |
| Severity    | High  |
| Description | <p>Whenever users deposit or withdraw tokens, the current exchange rate via the <code>_convertPOLToMaticX</code> / <code>_convertMaticXToPOL</code> functions is used to determine how much MaticX will be received for staking POL tokens or how much POL tokens are received for redeeming MaticX.</p> <p>This exchange rate is dependent on the circulating MaticX supply and the total staked POL amount:</p> <pre> function _convertPOLToMaticX( uint256 _balance ) private view returns (uint256, uint256, uint256) { uint256 totalShares = totalSupply(); totalShares = totalShares == 0 ? 1 : totalShares;  uint256 totalPooledAmount = getTotalStakeAcrossAllValidators(); if (totalPooledAmount == 0) { totalPooledAmount = 1; }  uint256 balanceInMaticX = (_balance * totalShares) / totalPooledAmount;  return (balanceInMaticX, totalShares, totalPooledAmount); }  function _convertMaticXToPOL( uint256 _balance ) private view returns (uint256, uint256, uint256) { uint256 totalShares = totalSupply(); totalShares = totalShares == 0 ? 1 : totalShares; </pre> |

```
uint256 totalPooledAmount = getTotalStakeAcrossAllValidators();
if (totalPooledAmount == 0) {
    totalPooledAmount = 1;
}
```

```
uint256 balanceInPOL = (_balance * (totalPooledAmount)) /
totalShares;
```

```
return (balanceInPOL, totalShares, totalPooledAmount);
}
```

An important invariant is that the exchange rate is not manipulated whenever deposits or withdrawals are happening but it is changed whenever the `stakeRewardsAndDistributeFees` function is invoked as this will increase the total staked **POL** amount without minting any **MaticX** tokens.

Due to the fact that the `stakeRewardsAndDistributeFees` function is not called before any deposit, users can trivially extract value from the protocol by depositing with the old exchange rate, then waiting for the **BOT** calling `stakeRewardsAndDistributeFees` which increases the exchange rate and requesting a withdrawal again.



In the current implementation, the interval in which the `stakeRewardsAndDistributeFees` function is called is not regular enough which means that one can steal fees which have been accrued since up to 3 days:

Latest 25 from a total of 720 transactions

| Transaction Hash                 | Method          | Block    | Age        | From                  | To                    | Amount | Txn Fee    |
|----------------------------------|-----------------|----------|------------|-----------------------|-----------------------|--------|------------|
| <a href="#">0x31a6c00184...</a>  | Stake Reward... | 20891123 | 9 hrs ago  | 0xF9d5f52C..81648D406 | Stader Labs: Matic... | 0 ETH  | 0.0031172  |
| <a href="#">0x991ccfb510...</a>  | Withdraw Val... | 20890525 | 11 hrs ago | 0xF9d5f52C..81648D406 | Stader Labs: Matic... | 0 ETH  | 0.00136815 |
| <a href="#">0xbfccc10be2c...</a> | Withdraw Val... | 20890525 | 11 hrs ago | 0xF9d5f52C..81648D406 | Stader Labs: Matic... | 0 ETH  | 0.00361829 |
| <a href="#">0x7ed4fa0348...</a>  | Stake Reward... | 20869611 | 3 days ago | 0xF9d5f52C..81648D406 | Stader Labs: Matic... | 0 ETH  | 0.00754865 |
| <a href="#">0xd70b09f53b3...</a> | Withdraw Val... | 20869013 | 3 days ago | 0xF9d5f52C..81648D406 | Stader Labs: Matic... | 0 ETH  | 0.00358858 |
| <a href="#">0x51f11ce26b4...</a> | Withdraw Val... | 20869013 | 3 days ago | 0xF9d5f52C..81648D406 | Stader Labs: Matic... | 0 ETH  | 0.00949056 |
| <a href="#">0x5d3d351719...</a>  | Stake Reward... | 20840924 | 7 days ago | 0xF9d5f52C..81648D406 | Stader Labs: Matic... | 0 ETH  | 0.00693137 |
| <a href="#">0x79723064bf9...</a> | Withdraw Val... | 20840327 | 7 days ago | 0xF9d5f52C..81648D406 | Stader Labs: Matic... | 0 ETH  | 0.00272713 |

which makes this blunder an easy target for malicious users to effortlessly extract value from the protocol.

At the time of writing, the following amount of rewards is just sitting uncompounded in the contract:

| ERC-20 Tokens (16)   |                       |
|--|-----------------------|
|  Polygon Ecos... (POL)<br>28,882.49740321 POL | \$10,599.13<br>@0.367 |
|  Matic Token (MATIC)<br>16,467.60328551 MATIC | \$6,089.37<br>@0.3698 |

This issue can be amplified if purchases are disallowed for certain periods (check: “Owner of *validatorShare* contract has several privileges that can negatively impact *MaticX*”) because this would mean the *stakeRewardsAndDistributeFees* function call will revert for some unknown period, resulting in even more tokens being accrued before applied to the exchange rate.

PoC:

1. The contract has 50k **POL** staked and 50k shares of *MaticX*, making the exchange rate 1:1 (1 share = 1 **POL**).
2. Over a few days, the contract accrues 5k **POL** in rewards.
3. A malicious user sees this and deposits 50K **POL**, receiving 50K new shares.
  - Now, the contract has 100K **POL** staked and 100K shares.
4. Later, *stakeRewardsAndDistributeFees* is called by the bot to stake the rewards, adding the 5K **POL** to the total.
  - The contract now has 105K **POL**, but still only 100K shares, making the exchange rate 1.05.
5. The attacker withdraws their 50K shares and receives 52.5K **POL** (because of the new exchange rate).

|                              |  |
|------------------------------|--|
|                              | <p>6. The attacker profits 2,500 POL (52,500 withdrawn - 50,000 deposited).</p> <p><b>Note:</b> In the PoC, the amounts have been simplified for the sake of clarity but a sophisticated attacker can set up a bot on-chain to execute this attack constantly and steal part of the yield meant for regular users.</p>   |
| <b>Recommendations</b>       | <p>Recommendations:</p> <p>Option 1: Consider invoking <code>withdrawRewards</code> (within a special iof-clause due to the <code>minAmount</code> requirement within <code>ValidatorShare</code>) and subsequently <code>stakeRewardsAndDistributeFees</code> (with the preferred deposit validator) to ensure that the <code>exchangeRate</code> is always up to date.</p> <p>Option 2: Consider incorporating the <code>ValidatorShare</code>'s native <code>restake</code> function and trigger it on every deposit/redeem whenever the <code>minAmount</code> threshold is met.</p> <p>Option 3: Consider ensuring that the BOT invokes <code>stakeRewardsAndDistributeFees</code> regularly (every 3 hours as example), using tools like Chainlink Automation.</p> <p>We recommend going with Option 3 as this does not further modify the codebase (which prevents the introduction of undesired side-effects).</p> |
| <b>Comments / Resolution</b> |  |

|             |   |
|-------------|---|
| Issue_07    | <b>MaticX</b> conversion is susceptible to inflation attack (after new deployment)  |
| Severity    | High  |
| Description | <p>The <b>MaticX</b> contract is susceptible to the standard inflation attack which is widely known by an unconsidered edge-case.</p> <p>The vault inflation attack means that the exchange rate for deposits is manipulated such that users will receive 0 shares (or a share amount which is rounded down) for their <b>POL/MATIC</b> deposits, which will then result in the previous depositor receiving all / the majority of deposits.</p> <p>The standard vault inflation attack is by simply depositing tokens and then donating ERC20 tokens to increase the underlying staked balance. This does however not work for <b>MaticX</b> as the exchange rate is not dependent on the ERC20 balance. Instead, there are two different scenarios of how this can be exploited:</p> <p><b>First scenario (theoretical):</b></p> <ul style="list-style-type: none"> <li>a) Depositing 1 wei of <b>POL/MATIC</b> (first depositor)</li> <li>b) Waiting until rewards are accrued and any address with the BOT role calls <b>stakeRewardsAndDistributeFees</b></li> <li>c) The exchange rate is now successfully manipulated</li> </ul> <p>This scenario is rather theoretical than practical because the 1 WEI deposit will likely not accrue any real rewards</p> <p><b>Second scenario (practical):</b></p> <ul style="list-style-type: none"> <li>a) Depositing 1 wei of <b>POL/MATIC</b> (first depositor)</li> <li>b) Delegating funds from the own address to a <b>ValidatorShare</b> contract</li> <li>c) Transferring the <b>ValidatorShare</b> token directly to the <b>MaticX</b> contract</li> <li>d) The exchange rate is now manipulated</li> </ul> |



PoC:

To run this PoC, paste the test in the file **MaticX.spec.t.ts**

```
it("Inflation attack", async function () {
    const { maticX, matic, stakerA, stakerB, stakeManager,
    preferredDepositValidatorId } = await loadFixture(deployFixture);

    // Staker A submits 1 wei of MATIC
    await matic.connect(stakerA).approve(maticX.address,
    ethers.constants.MaxUint256);
    await maticX.connect(stakerA).submit(1);

    // Staker A directly stakes 1e18 MATIC to the validator
    await matic.connect(stakerA).approve(stakeManager.address,
    ethers.constants.MaxUint256);
    const validatorShareAddress = await
    stakeManager.getValidatorContract(preferredDepositValidatorId);
    const validatorShare = await
    ethers.getContractAt("ValidatorShare", validatorShareAddress);
    const stakeAmount = ethers.utils.parseUnits("1", 18);
    await validatorShare.connect(stakerA).buyVoucher(stakeAmount,
    0);

    // Staker A transfers the 1e18 shares of validator to MaticX
    await validatorShare.connect(stakerA).transfer(maticX.address,
    stakeAmount);

    // The exchange rate now is inflated (1:1e18+1)
    const exchangeRate = await maticX.convertMaticXToPOL(1);
    expect(exchangeRate[0]).to.equal(ethers.BigNumber.from("1000
    0000000000000001"));

    // Now, staker B stakes 1e18 MATIC
    await matic.connect(stakerB).approve(maticX.address,
    ethers.constants.MaxUint256);
    await maticX.connect(stakerB).submit(stakeAmount);

    // Now we check the shares of staker A and staker B
    const sharesA = await maticX.balanceOf(stakerA.address);
    const sharesB = await maticX.balanceOf(stakerB.address);
```


|                              |  |
|------------------------------|--|
|                              | <pre>// Staker A has stolen the funds from staker B expect(sharesA).to.equal(ethers.BigNumber.from("1")); expect(sharesB).to.equal(ethers.BigNumber.from("0")); });</pre> <p>Additionally, a malicious user can also frontrun the first deposit by purchasing <b>ValidatorShare</b> tokens and transferring them to the <b>MaticX</b> contract which breaks the ratio.</p>   |
| <b>Recommendations</b>       | <p>This exploit only works for the first depositor, since the current on-chain deployment already has a healthy supply distribution, it is impossible to execute this exploit.</p> <p>For future deployments, there are several options to prevent this:</p> <ul style="list-style-type: none"> <li>a) Include a <b>minAmountOut</b> parameter</li> <li>b) Transfer 1000 shares to <b>Oxdead</b> during the first deposit</li> <li>c) Prevent the scenario where zero shares are received</li> <li>d) Executing the first deposit after the deployment</li> </ul> <p>We do not recommend updating the proxy implementation with a fix because this will introduce unnecessary risk. However, in the future this should be definitely fixed whenever the contract is newly deployed. Additionally we recommend governance to execute a small deposit that, in such a scenario where all <b>MaticX</b> tokens have been redeemed, there is still a small amount allocated to governance which prevents the init-state.</p> |
| <b>Comments / Resolution</b> |  |

|             |   |
|-------------|---|
| Issue_08    | Sophisticated exploit allows users to permanently lock all rewards into the <b>MaticX</b> contract  |
| Severity    | High  |
| Description | <p>The <b>ValidatorShare</b> contract exposes a <b>transfer</b> function which automatically “claims” rewards from the “from” and “to” address. Optionally this can be done by claiming <b>POL</b> or <b>MATIC</b>:</p> <pre> function transfer(address to, uint256 value) public returns (bool) {     _transfer(to, value, false);     return true; }  function _transfer(address to, uint256 value, bool pol) internal {     address from = msg.sender;     // get rewards for recipient     _withdrawAndTransferReward(to, pol);     // convert rewards to shares     _withdrawAndTransferReward(from, pol);     // move shares to recipient     super._transfer(from, to, value);     _getOrCacheEventsHub().logSharesTransfer(validatorId, from, to, value); } </pre> <p>A malicious user can first buy a small amount of <b>ValidatorShare</b> tokens and then transfer these ValidatorShare tokens to the <b>MaticX</b> contract via the <b>transfer</b> function, which will then claim all <b>MATIC</b> tokens instead of <b>POL</b> tokens.</p> <p>These tokens will be locked because there is no way to withdraw /allocate them as reward tokens, as the <b>stakeRewardsAndDistributeFees</b> function only handles <b>POL</b> tokens and no <b>MATIC</b> tokens.</p> <p>PoC:</p> |

|                              |  |
|------------------------------|--|
|                              | <ol style="list-style-type: none"> <li>1. The <b>MaticX</b> contract accrues some rewards on the validators over time.</li> <li>2. An attacker realizes that and decides to stake a dust amount of funds directly into the validators that have pending rewards to distribute.</li> <li>3. The attacker then transfers the shares directly to the <b>MaticX</b> contract. <ul style="list-style-type: none"> <li>• The function <b>ValidatorShare::transfer</b> automatically claims rewards in <b>MATIC</b> for the sender and receiver of the tokens.</li> <li>• Therefore, the <b>MaticX</b> contract will receive the accrued rewards in <b>MATIC</b> tokens</li> </ul> </li> <li>4. Because <b>stakeRewardsAndDistributeFees</b> only stakes <b>POL</b> and not <b>MATIC</b>, the rewards in <b>MATIC</b> will be stuck in the contract.</li> </ol> |
| <b>Recommendations</b>       | Consider adjusting the <b>stakeRewardsAndDistributeFees</b> function to be compatible with <b>MATIC</b> tokens as well.  |
| <b>Comments / Resolution</b> |  |

| Issue_09    | Exotic edge-case will result in bricked <code>_submit</code> function   |
|-------------|---|
| Severity    | High  |
| Description | <p>A very exotic edge-case can result in a scenario where deposits are bricked and users will always lose all deposited <b>POL/MATIC</b> tokens.</p> <p>Consider a scenario where the <b>MaticX</b> contract is successfully operating for some time and at some point all users decide to redeem their <b>MaticX</b> for the underlying <b>POL</b> tokens. (This can be either all at once due to a specific event or step by step)</p> <p>An event could for example be the scenario if the usecase of <b>MaticX</b> is non-existent anymore (temporarily) because there are no yield sources where users can stake <b>MaticX</b> or rewards are paused for <b>POL</b> staking. However, unrelated to the reason for such a scenario, it is definitely possible for it to happen.</p> <p>Once the last user redeems their <b>MaticX</b> for <b>POL</b> tokens, that means the circulating <b>MaticX</b> supply becomes zero. But eventually there will be some idle rewards sitting in the contract. The <b>BOT</b> now automatically restakes these rewards and the contract is suddenly in the state where there are staked <b>POL</b> tokens without any circulating <b>MaticX</b> supply.</p> <p>If a new user deposits, the following calculation for <b>POL</b> -&gt; <b>MaticX</b> is happening:</p> <pre>uint256 balanceInMaticX = (_balance * totalShares) / totalPooledAmount;</pre> <p>depending on the provided <code>_balance</code> and the <code>totalPooledAmount</code> (which was increased due to the restake), it can happen that the user will not receive any <b>MaticX</b> tokens and the provided <b>POL</b> tokens will be lost.</p> |

|                              |  |
|------------------------------|--|
|                              | <p>PoC:</p> <ul style="list-style-type: none"> <li>- <b>MaticX</b> supply = 0</li> <li>- <b>totalPooledAmount</b> = 1000e18 (due to <b>BOT</b> restaking)</li> </ul> <p>a) Alice deposits 100e18 <b>POL</b> tokens</p> <p><math>&gt; (100e18 * 1) / 1000e18 = 0</math></p> <p>b) Alice does not receive any <b>MaticX</b> tokens in exchange</p> |
| <b>Recommendations</b>       | <p>Consider simply depositing a reasonable amount into the <b>MaticX</b> contract as governance or independent third-party which will never be withdrawn. This will ensure a scenario with zero circulating <b>MaticX</b> tokens and idle rewards can never happen.</p>  |
| <b>Comments / Resolution</b> |  |

|                       |   |
|-----------------------|---|
| Issue_10              | Initial <b>MATIC</b> balance will be stuck in the contract after upgrade  |
| Severity              | High  |
| Description           | <p>The current on-chain implementation has an idle amount of <b>MATIC</b> tokens, at the time of writing it is the following amount:</p> <div data-bbox="512 636 1401 745" data-label="Complex-Block">  <div> <div>Matic Token (MATIC)</div> <div>16,467.60328551 MATIC</div> </div> <div> <div>\$6,089.37</div> <div>@0.3698</div> </div> </div> <p>If the proxy is now upgraded without these rewards being compounded beforehand, they will be stuck in the contract because the new implementation is incompatible with <b>MATIC</b> tokens.</p> |
| Recommendations       | Consider adjusting the <code>stakeRewardsAndDistributeFees</code> function to be compatible with <b>MATIC</b> tokens as well and consider compounding these idle rewards before the upgrade.  |
| Comments / Resolution |   |

|             |   |
|-------------|---|
| Issue_11    | Malicious user can DoS withdrawals by dusting the <b>MaticX</b> contract with small amounts of <b>ValidatorShare</b> tokens from different validators   |
| Severity    | High  |
| Description | <p>The <b>requestWithdrawal</b> function loops over all existing validators until either <b>leftAmountToWithdraw</b> = 0 or until all validators have been considered.</p> <p>In theory, this exposes an issue where the loop runs OOG at some point. Due to the fact that the architecture exposes a <b>preferred depositor</b>, this attack cannot be executed as one cannot deposit 1 wei to different validators to trigger a scenario where one validator has an insufficient balance to cover a withdrawal while it then loops over x amount of validators which have all a balance of 1 wei.</p> <p>However, we still found a way to execute this exploit. That being said, there are a two prerequisites that are needed:</p> <ul style="list-style-type: none"> <li>a) The architecture must expose a large amount of different existing validators</li> <li>b) A withdrawal attempt must result in one validator being depleted which triggers the loop continuation to other validators</li> </ul> <p>Prerequisite a) is currently <b>NOT GIVEN</b> based on the on-chain implementation. <b>This means this issue can only happen once more validators are being added.</b></p> <p>Once these prerequisites are given, a user can trivially buy shares from different validators, transfer them to the <b>MaticX</b> contract which will then result in <b>amountToWithdrawFromValidator</b> &gt; 0 and attempts to loop over all validators which will potentially run OOG.</p> <p>PoC:</p> <ol style="list-style-type: none"> <li>1. The registry contains a ton of validators, and most of them are not currently used by the <b>MaticX</b> contract.</li> </ol> |



|                              |  |
|------------------------------|--|
|                              | <ol style="list-style-type: none"> <li>2. An attacker sees that and decides to stake a dust amount of <b>POL</b> in all empty validators.</li> <li>3. The attacker directly transfers all those validators' shares to the <b>MaticX</b> contract.</li> <li>4. A regular user tries to make a big withdrawal that depletes the preferred validator for withdrawals, and the function has to loop over most of the registered validators due to the dust amounts of shares.</li> <li>5. Because the function <b>sellVoucher_newPOL</b> is gas-intensive within each validator, the transaction will possibly run out of gas trying to withdraw the dust amounts of <b>POL</b> from a ton of validators.</li> <li>6. The withdrawal attempt reverts.</li> </ol> |
| <b>Recommendations</b>       | Consider ensuring that only a reasonable amount of validators exist in the registry.   |
| <b>Comments / Resolution</b> |  |

|                       |   |
|-----------------------|---|
| Issue_12              | <code>feePercent</code> change will be applied in hindsight   |
| Severity              | Medium  |
| Description           | The <code>setFeePercent</code> function allows for changing the fee which is taken upon reward distribution. A change of this fee will be applied in hindsight on the current existing ERC20 balance in the contract, changing the expected reward distribution from already accrued rewards. |
| Recommendations       | Consider invoking <code>stakeRewardsAndDistributeFees</code> before any fee change. (If there any any idle rewards)   |
| Comments / Resolution |   |

|             |  |
|-------------|--|
| Issue_13    | Cached <code>requestEpoch</code> during <code>requestWithdraw</code> will be inaccurate if <code>WITHDRAWAL_DELAY</code> is changed after a withdrawal has been requested  |
| Severity    | Medium   |
| Description | <p>A blunder within the <code>requestWithdraw</code> function will potentially disallow users to rightfully claim their withdrawal on time:</p> <pre>uint256 requestEpoch = stakeManager.epoch() + stakeManager.withdrawalDelay();</pre> <p>The <code>requestEpoch</code> is determined by using the current <code>withdrawalDelay()</code> at the time of requesting the withdrawal.</p> <p>This is incorrect due to the fact that the check within the <code>ValidatorShare</code> contract is as follows:</p> <pre>require(</pre> |

|                              |  |
|------------------------------|--|
|                              | <pre> unbond.withdrawEpoch.add(stakeManager.withdrawalDelay()) &lt;= stakeManager.epoch() &amp;&amp; shares &gt; 0,     "Incomplete withdrawal period" ); </pre> <p>which is using the dynamic <code>withdrawalDelay()</code> value while the <code>claimWithdrawal</code> function uses <code>requestEpoch</code> which is corresponding to the <code>withdrawalDelay</code> at the time of the request creation.</p> <p>If the <code>WITHDRAWAL_DELAY</code> value is now decreased after a request has been made, users should theoretically be able to claim their request earlier (as per code within <code>ValidatorShare</code>). However, due to the blunder within the <code>requestWithdraw</code> function, this is impossible.</p> |
| <b>Recommendations</b>       | <p>Consider following the same approach as the <code>ValidatorShare</code> contract by storing the <code>currentEpoch</code> into the <code>WithdrawalRequest</code> and applying the dynamic <code>withdrawalDelay()</code> on the check.</p> <p>Optionally, one can simply remove the epoch check within the <code>MaticX</code> contract as it would revert anyways within the <code>ValidatorShare</code> contract if the epoch for the nonce is not reached.</p>  |
| <b>Comments / Resolution</b> |  |

|                              |  |
|------------------------------|--|
| <b>Issue_14</b>              | Rare possibility of DoS'ing withdrawals by allocating dust reward amounts to many different validators in case of malicious <b>BOT</b> address   |
| <b>Severity</b>              | <b>Low</b>   |
| <b>Description</b>           | <p>This issue is similar to the “Malicious user can DoS withdrawals by dusting the MaticX contract with small amounts of ValidatorShare tokens from different validators” issue.</p> <p>However, the root-cause of this issue is the fact that the <b>BOT</b> address can delegate funds to any validator via the <b>stakeRewardsAndDistributeFees</b> function (instead of only to the preferred depositor)</p> |
| <b>Recommendations</b>       | <p>Consider strictly ensuring that:</p> <ul style="list-style-type: none"> <li>a) No unnecessary large amount of validators is listed within the <b>ValidatorRegistry</b> contract</li> <li>b) The <b>BOT</b> address is a (partially) trusted address</li> </ul>  |
| <b>Comments / Resolution</b> |  |

| Issue_15   |   |
|--|---|
| Integration Issue: Enforcement of POL instead of MATIC |   |
| Severity   | Low   |
| Description  | Currently, the <code>claimWithdrawal</code> function only transfers out POL and does not allow for choosing whether POL/MATIC should be used. This could result in issues for protocols that are building on top of MaticX as they now essentially need to adjust their logic to handle POL instead of MATIC. |
| Recommendations  | We do not recommend a change. However, it should be communicated with protocols which are built on top of MaticX.   |
| Comments / Resolution                                  |   |

| Issue_16  |  |
|---|--|
| Lack of reasonable upper limit for <code>setFeePercent</code> |  |
| Severity  | Low  |
| Description   | <p>The <code>setFeePercent</code> function allows setting the treasury fee of up to 100%.</p> <p>This amount is unreasonably high, as this means all fees would completely go towards the protocol and users would not receive any fee at all.</p> |
| Recommendations   | Consider changing this to a reasonable threshold (e.g. 10%)  |
| Comments / Resolution   |  |

|                       |  |
|-----------------------|--|
| Issue_17              | Trigger of temporary DoS of withdrawals due to validator unavailability by malicious actor   |
| Severity              | Low  |
| Description           | <p>Within the “Governance of <i>ValidatorShare</i> and <i>StakeManager</i> contracts has several privileges that can negatively impact <i>MaticX</i>” issue, we have explained that under several circumstances it can happen that the contract does not work as expected due to some changes within the <i>ValidatorShare</i> or <i>StakeManager</i> contract. One explicit scenario is the scenario where the <i>deactivationEpoch</i> of a validator is above the current epoch:</p> <pre> else if (deactivationEpoch &gt; currentEpoch) { // validator just unstaked, need to wait till next checkpoint     revert("unstaking"); } </pre> <p>which is the case (as the comment mentions) whenever a validator has just unstaked.</p> <p>In the scenario where there are no delegated stakes towards this validator, withdrawals will always work.</p> <p>However if a malicious user recognizes such a transaction by the validator and frontruns this with a dust purchase and transfer towards the <i>MaticX</i> contract, it may happen that this validator would then be part of the <i>requestWithdraw</i> loop which would then revert due to the above mentioned issue.</p> |
| Recommendations       | We do not recommend a change. However, it must be kept in mind that such a scenario can be intentionally triggered by a malicious user and other users are forced to wait with their withdrawals.  |
| Comments / Resolution |  |

| Issue_18  |  |
|---|--|
| Reentrancy guard is only initialized during <code>initializeV2</code> |  |
| <b>Severity</b>   | <b>Informational</b>   |
| <b>Description</b>  | <p>The reentrancy guard is corresponding to values 1 and 2 for <code>ENTERED</code> and <code>NOT_ENTERED</code>:</p> <pre>uint256 private constant NOT_ENTERED = 1; uint256 private constant ENTERED = 2;</pre> <p>By default however, the value is zero:</p> <pre>uint256 private reentrancyGuardStatus;</pre> <p>which means that the very first function call will not be guarded with the reentrancy guard, if <code>initializeV2</code> is not invoked beforehand. (after the first function call it is set to <code>NOT_ENTERED</code>)</p> |
| <b>Recommendations</b>  | <p>Consider immediately calling <code>initializeV2</code> after the proxy upgrade. Since this contract also functions without <code>initializeV2</code>, it is possible for users to interact with the contract immediately after the upgrade, before <code>initializeV2</code> is called. (If the proxy upgrade and <code>initializeV2</code> call are not in the same transaction).</p> <p>Optionally, one can mark it as <code>NOT_ENTERED</code> by default in the storage declaration.</p>  |
| <b>Comments / Resolution</b>  |  |

|                       |   |
|-----------------------|---|
| Issue_19              | Griefing: <code>requestWithdraw</code> before <code>migrateDelegation</code> can prevent migration  |
| Severity              | Informational   |
| Description           | Whenever the <code>migrateDelegation</code> function is called with the full existing balance of a validator , a user can simply invoke <code>requestWithdraw</code> with 1 wei beforehand which would then result in a revert of the <code>migrateDelegation</code> function because <code>_amount</code> is larger than the existing balance. Notably, it must be the preferred deposit/withdrawal validator. |
| Recommendations       | We do not see the necessity of a change. However, if still desired to fix one can simply cross-check the staked owned balance for the specific validator and downsize the <code>_amount</code> parameter to match the balance.  |
| Comments / Resolution |   |



| Issue_20  |  |
|---|--|
| Griefing: Prevention of <code>withdrawValidatorsReward</code> |  |
| Severity  | Informational  |
| Description   | <p>The <code>withdrawValidatorsReward</code> function allows for claiming rewards from multiple different validators within the same transaction. This function is vulnerable to griefing because a user can simply invoke the <code>withdrawRewards</code> function to withdraw rewards from one validator in the parameter list which then results in a revert of the <code>withdrawValidatorsReward</code> function call due to the following check within the <code>ValidatorShare</code> contract:</p> <pre>require(rewards &gt;= minAmount, "Too small rewards amount");</pre> |
| Recommendations   | We do not see the necessity of a change. However, this should be kept in mind.   |
| Comments / Resolution   |  |

| Issue_21                                       |   |
|--|---|
| Treasury fee granularity might be insufficient |   |
| Severity                                       | Informational   |
| Description                                    | The treasury fee can be set between 0 and 100. The current setup lacks granularity in scenarios where it is desired to for example set a fee of 4.5%. |
| Recommendations                                | Consider if it is ever desired to increase the granularity. If yes, consider increasing the fee calculation to use BPS of 10_000.                     |
| Comments / Resolution                          |   |

|                              |   |
|------------------------------|---|
| <b>Issue_22</b>              | <code>_submit</code> without preferred depositor being set will always result in using <code>validatorId = 0</code>   |
| <b>Severity</b>              | <b>Informational</b>  |
| <b>Description</b>           | <p>The <code>_submit</code> function fetches the preferred depositor as follows:</p> <pre>uint256 preferredValidatorId = validatorRegistry     .preferredDepositValidatorId(); IValidatorShare validatorShare = IValidatorShare(     stakeManager.getValidatorContract(preferredValidatorId)     );</pre> <p>If the preferred depositor is not set, this will always return zero, fetching the preferred validator with the ID = 0 (due to <code>uint256</code> being by default 0).</p> <p>Fortunately, the validator with ID = 0 is not set within the <code>StakeManager</code> and always corresponds to <code>address(0)</code> which results in a revert.</p> |
| <b>Recommendations</b>       | We do not recommend a change. However, if still desired to fix this, consider simply reverting directly if the preferred depositor returns ID = 0.  |
| <b>Comments / Resolution</b> |   |

|                       |   |
|-----------------------|---|
| Issue_23              | Setting <b>DEFAULT_ADMIN_ROLE</b> as <b>roleAdmin</b> for BOT role is redundant   |
| Severity              | Informational   |
| Description           | <p>The <b>DEFAULT_ADMIN_ROLE</b> has by default all privileges to add/revoke roles due to the function returning 0x00 if no <b>roleAdmin</b> is set:</p> <pre> /**  * @dev Returns the admin role that controls `role`. See  {grantRole} and  * {revokeRole}.  *  * To change a role's admin, use {_setRoleAdmin}.  */ function getRoleAdmin(bytes32 role) public view override returns (bytes32) {     return _roles[role].adminRole; } </pre> <p>Therefore, it is not necessary to set it as role admin for the <b>BOT</b> role. If however, in the previous implementation a different <b>roleAdmin</b> has been set for the <b>BOT</b> role, this means that the <b>DEFAULT_ADMIN_ROLE</b> is no longer the <b>roleAdmin</b> and therefore a change is necessary to restore this state.</p> |
| Recommendations       | Consider thinking about if this change is necessary. Additionally we recommend adding tests to ensure the <b>DEFAULT_ADMIN_ROLE</b> is in fact the <b>roleAdmin</b> (which is the default case if nothing has been changed).  |
| Comments / Resolution |   |

|                 |   |
|-----------------|---|
| Issue_24        | Arithmetic operations within <code>ValidatorShare._buyShares</code> can result in some dusted <code>MATIC</code> within <code>MaticX</code> contract  |
| Severity        | Informational   |
| Description     | <p>The <code>MaticX</code> contract allows users to deposit <code>POL</code> and <code>MATIC</code> tokens. The ERC20 flow is as follows:</p> <ul style="list-style-type: none"> <li>a) Transfer <code>MATIC</code> from caller to <code>MaticX</code> contract</li> <li>b) Transfer <code>MATIC</code> from <code>MaticX</code> contract to <code>StakeManager</code> contract</li> </ul> <p>The arithmetic operations within <code>ValidatorShare._buyShares</code> are as follows:</p> <pre>uint256 shares = _amount.mul(precision).div(rate); // clamp amount of tokens in case resulted shares requires less tokens than anticipated _amount = rate.mul(shares).div(precision);</pre> <p>As one can already identify from the <b>comment</b>, it may be possible that <code>_amount</code> is less than the <code>_amount</code> parameter provided.</p> <p>This means if users want to stake <code>MATIC</code> instead of <code>POL</code> that eventually not all <code>MATIC</code> is being transferred to the <code>StakeManager</code> contract which results in some dust <code>MATIC</code> being locked within the <code>MaticX</code> contract.</p> <p>During our inspection, we could not identify such a scenario to happen in the current implementation (because slashing is disabled which means that the rate is always 100 or 1e29)</p> <p>However, given the existence of this comment, we are still of the opinion to raise this issue for eventual future upgrades.</p> |
| Recommendations | Consider simply adjusting the <code>stakeRewardAndDistributeFees</code> function to also enable <code>MATIC</code> .  |

|                       |  |
|-----------------------|--|
| Comments / Resolution |  |
|-----------------------|--|

|             |   |
|-------------|---|
| Issue_25    | Future Upgrade: Slashing after withdrawal request will result in incorrect output amount  |
| Severity    | Informational   |
| Description | <p>Currently, the <code>StakeManager</code> contract does not allow for slashing. However, in the future it might be very much possible that this feature is introduced, which can result in a decrease of the <code>withdrawExchangeRate</code> in the corresponding <code>ValidatorShare</code> contract.</p> <p>When inspecting the overall business logic within the <code>MaticX</code> contract, users can always gauge how much <code>POL</code> tokens they will receive for the provided amount of <code>MaticX</code> using the following calculation:</p> $\text{maticX} * \text{totalPol} / \text{totalMaticX}$ <p>This is also displayed within the <code>convertMaticXToPol</code> function.</p> <p>However, the Polygon staking architecture uses a slightly different calculation when transferring out <code>POL</code> tokens during the <code>unstakeClaimTokens_newPOL</code> function.</p> <p>A pool and share based approach is used which transfers out <code>POL</code> tokens based on the so-called <code>withdrawExchangeRate()</code>. This logic commingles all requested funds with their initial exchange rate and pays out the average rate. This means that users which have requested earlier might get more tokens than expected and users which have requested later might get less tokens. (In the scenario of a slashing event)</p> |

## PoC:

### Status Quo:

The status quo is that users have deposited in **MaticX** which mints **MaticX** tokens to users and deposits into the **ValidatorShare** contract which grants **ValidatorShare** tokens to the **MaticX** contract.

- maticXSupply = 1000e18
- balanceValidatorShares = 1000e18
- exchangeRate = 100
- Alice has 500e18 **MaticX** token and Bob has 500e18 **MaticX** token

a) Alice calls **requestWithdraw** which burns 500e18 **MaticX** tokens. Alice expects to get 500e18 POL tokens out

- > withdrawPool = 500e18
- > withdrawShares = 500e18
- > withdrawExchangeRate = 100
- > unbond.shares = 500e18

b) The validator is slashed which will decrease the exchangeRate to 80

c) Bob calls **requestWithdraw** which burns 500e18 **MaticX** tokens. Bob expects to get 400e18 POL tokens out

- > withdrawPool = 900e18
- > withdrawShares = 1000e18
- > withdrawExchangeRate = 90
- > unbond.shares = 500e18

d) Alice calls **claimWithdrawal**

- > Alice receives 450e18 **POL** instead of the expected 500e18 **POL**

|                              |  |
|------------------------------|--|
| <b>Recommendations</b>       | <p>We do not recommend a change as this is based on the underlying concept of the <a href="#">ValidatorShare</a> contract which will also be influenced by requests outside from <a href="#">MaticX</a>.</p> <p>A comment could be added which indicates that in such a scenario users may not receive the expected output amount.</p> |
| <b>Comments / Resolution</b> |  |

## Bonus

|             |   |
|-------------|---|
| Issue_26    | Arbitrage opportunity in <b>ChildPool</b> due to delayed update on exchange rate  |
| Severity    | High / Medium   |
| Description | <p>The <b>ChildPool</b> contract allows users to deposit <b>POL</b> and get <b>MaticX</b> at an exchange rate that is obtained from the <b>MaticX</b> contract in Mainnet.</p> <p>When the exchange rate in <b>MaticX</b> increases, it sends the new rate through the Polygon bridge to <b>ChildPool</b>. After some minutes of the update in Mainnet, the <b>ChildPool</b> in Polygon is updated with the new exchange rate.</p> <p>An attacker can use this feature to sandwich the rate updates in <b>ChildPool</b> and steal part of the yield that should go to regular users.</p> <p>The attack sequence would be the following:</p> <ol style="list-style-type: none"> <li>1. The rate increases in <b>MaticX</b> and the update is sent through the bridge to Polygon.</li> <li>2. An attacker sees that and buys a huge amount of shares in <b>ChildPool</b> (<code>swapMaticForMaticXViaInstantPool</code>).</li> <li>3. After some minutes, when the new rate is updated in <b>ChildPool</b>, the attacker requests a withdrawal (<code>requestMaticXSwap</code>).</li> <li>4. After the lock period has passed, the attacker will have the funds back with an extra profit from the exchange rate increase.</li> </ol> <p>Currently, this issue is partially fixed by a lockup period in <b>ChildPool</b> of 50 hours. However, the rate increase in <b>MaticX</b> usually happens every 3 days or more, therefore allowing an attacker to benefit from all the rate increases without actually having the funds deposited all the time in <b>ChildPool</b>.</p> |



|                              |   |
|------------------------------|---|
| <b>Recommendations</b>       | <p>To mitigate this issue is recommended to implement some of the following measures:</p> <ul style="list-style-type: none"><li>- Increase the regularity of the calls to <code>stakeRewardsAndDistributeFees</code>, so that the rate increase happens more often.</li><li>- Increase the value of the variable <code>maticXSwapLockPeriod</code> in <code>ChildPool</code> so that the funds have to be locked for a longer time in the contract before a withdrawal.</li></ul> |
| <b>Comments / Resolution</b> |   |