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	https://github.com/stader-labs/maticX/tree/6889ca1b630e294131660c83816078d74465a7a0/contracts
Resolution 1	https://github.com/stader-labs/maticX/tree/3f234e8cfb3c2f688b4b2346024a8a7eeb413b14/contracts

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	2	2		
Medium	5	3		2
Low	5	2		3
Informational	9	4		5
Governance	2			2
Total	23	11		12

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	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. One can simply use hardhat-storage-layout or forge layout storage CONTRACT
Recommendations	Consider keeping this in mind when upgrading
Comments / Resolution	Acknowledged

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 removeValidator by dusting a small amount of ValidatorShare tokens to the MaticX contract
Severity	**Medium**
Description	The removeValidator function allows for removing any validator which has a zero-balance and is not a preferred validator: *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");* A malicious user can simply purchase a small amount of the ValidatorShare token, transfer them to MaticX which then results in a revert of the following check: *require(validatorBalance == 0, "Validator has some shares left");* and essentially prevents the removal from validators.
Recommendations	Consider implementing a _ignoreBalance boolean which optionally allows to bypass this check in such scenarios. Optionally, it is also possible to execute a migration followed by a removeValidator call (in the same transaction) to remove any dusted

	validators or execute a temporary change of the MaticX address (which can also be dusted if not removed within the same transaction).
Comments / Resolution	Resolved.

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

MaticX

This section of the report only displays issues which are related to an existing deployment. Issues which are related to new deployments will be handled in the section below.

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:

> POLAmount * MaticXSupply / stakedPOL

> MaticXAmount * stakedPol / 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.

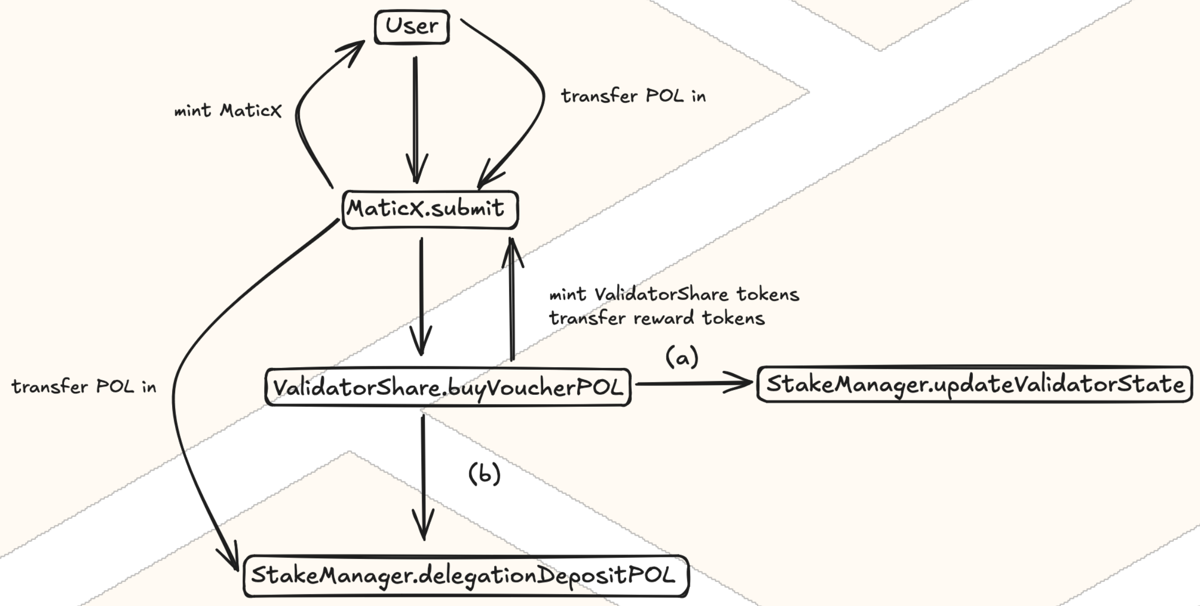
Furthermore this contract exposes several other functions but the above mentioned functions are the only ones used by the MaticX contract.

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.

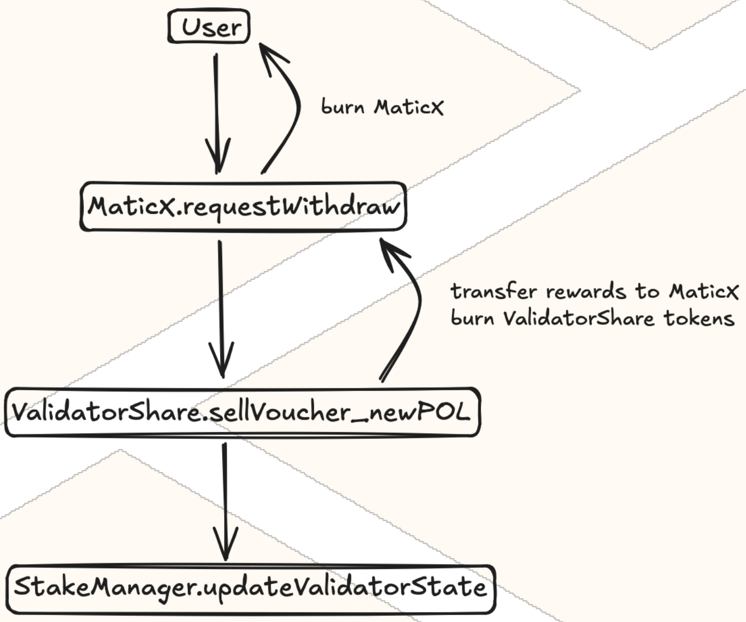
The flow is as follows:


User
mint MaticX
transfer POL in
MaticX.submit
mint ValidatorShare tokens
transfer reward tokens
transfer POL in
(a)
ValidatorShare.buyVoucherPOL
StakeManager.updateValidatorState
(b)
StakeManager.delegationDepositPOL

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.

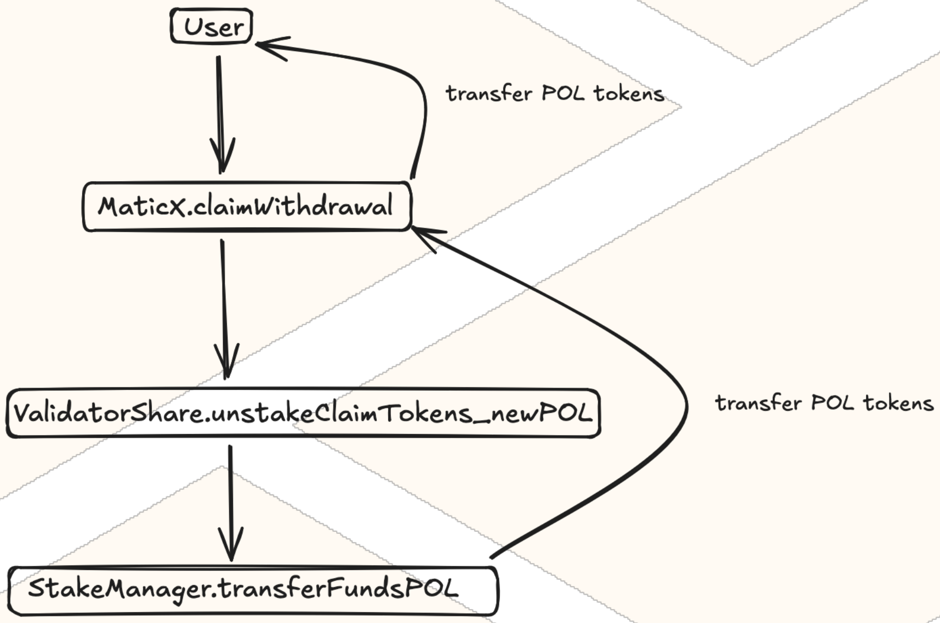
The flow is as follows:


User
burn MaticX
MaticX.requestWithdraw
transfer rewards to MaticX
burn ValidatorShare tokens
ValidatorShare.sellVoucher_newPOL
StakeManager.updateValidatorState

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:


User
transfer POL tokens
MaticX.claimWithdrawal
ValidatorShare.unstakeClaimTokens_newPOL
transfer POL tokens
StakeManager.transferFundsPOL

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.

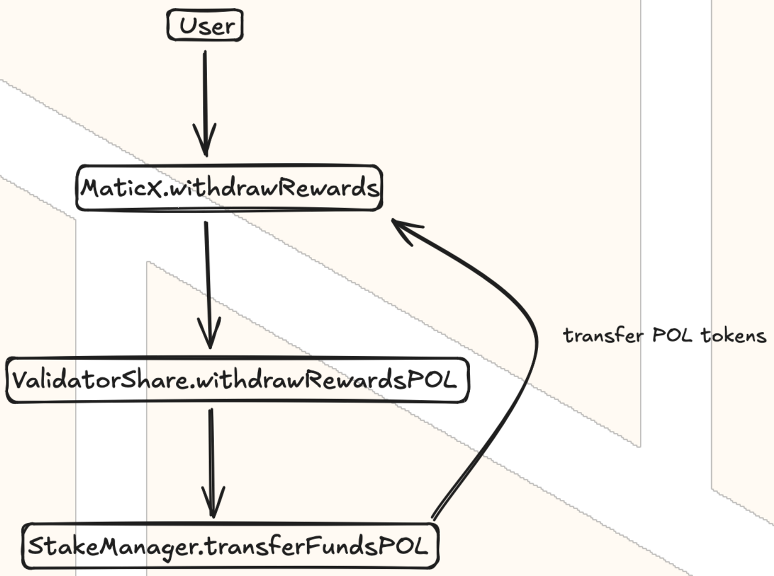
Appendix: Reward Mechanism

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.

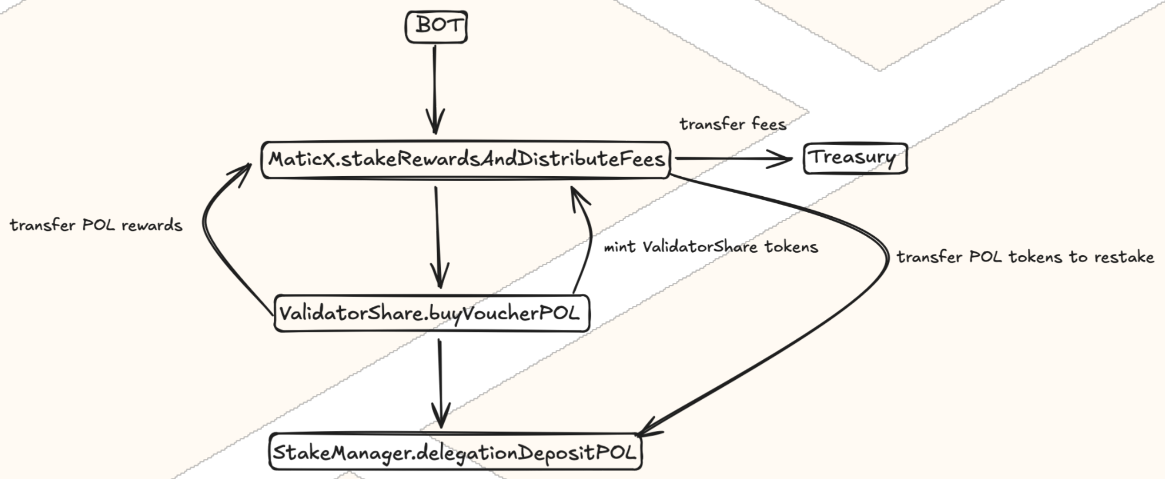
Rewards can be simply claimed by anyone via invoking withdrawRewards (for one validator) or withdrawValidatorsReward, which then claims rewards from the corresponding validator(s) via the withdrawRewardsPOL function and transfers these to the MaticX contract (this also automatically happens upon each 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:


User
MaticX.withdrawRewards
ValidatorShare.withdrawRewardsPOL
StakeManager.transferFundsPOL
transfer POL tokens

The flow for staking rewards is as follows:


BOT
MaticX.stakeRewardsAndDistributeFees
transfer fees
Treasury
transfer POL rewards
mint ValidatorShare tokens
transfer POL tokens to restake
ValidatorShare.buyVoucherPOL
StakeManager.delegationDepositPOL

Appendix: Core Invariants

The core invariants of the protocol are the following:

1) Deposits should always increase the totalMaticXSupply by

> depositAmount * totalMaticXSupply / stakedPol

2) Deposits should never influence the MaticX exchange rate

3) Withdrawals should always decrease stakedPol by

> maticXRedeemed * stakedPol / 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

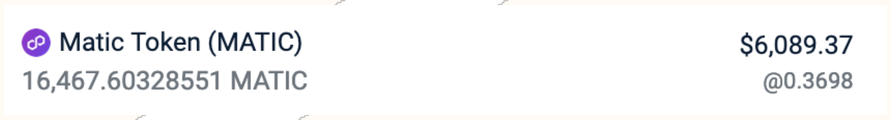
Issue_04	Governance Privilege: Contract owner has control over funds
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. This includes several functionalities such as pausing,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

Issue_05	Governance of ValidatorShare and StakeManager contracts has several privileges that can negatively impact MaticX
Severity	**Governance**
Description	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: For ValidatorShare: a) migrateOut b) migrateIn c) updateDelegation For StakeManager: a) setDelegationEnabled b) setStakingToken c) unstake/unstakePOL d) drain 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.
Recommendations	We do not recommend a change. We assume that the Polygon team is highly trusted.
Comments / Resolution	Acknowledged.

Issue_06	Sophisticated exploit allows users to permanently lock all rewards into the MaticX contract
Severity	**High**

Description	The ValidatorShare contract exposes a transfer function which automatically "claims" rewards from the "from" and "to" address. Optionally this can be done by claiming POL or MATIC:
	
	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);
	}
	
	A malicious user can first buy a small amount of ValidatorShare tokens and then transfer these ValidatorShare tokens to the MaticX contract via the transfer function, which will then claim all MATIC tokens instead of POL tokens.
	These tokens will be locked because there is no way to withdraw /allocate them as reward tokens, as the stakeRewardsAndDistributeFees function only handles POL tokens and no MATIC tokens.
	
	PoC:

	1. The MaticX contract accrues some rewards on the validators over time. 2. An attacker realizes that and decides to stake a dust amount of funds directly into the validators that have pending rewards to distribute. 3. The attacker then transfers the shares directly to the MaticX contract. • The function ValidatorShare::transfer automatically claims rewards in MATIC for the sender and receiver of the tokens. • Therefore, the MaticX contract will receive the accrued rewards in MATIC tokens 4. Because stakeRewardsAndDistributeFees only stakes POL and not MATIC, the rewards in MATIC will be stuck in the contract.
Recommendations	Consider adjusting the stakeRewardsAndDistributeFees function to be compatible with MATIC tokens as well.
Comments / Resolution	Resolved, a function has been implemented which allows for compounding MATIC tokens.

Issue_07	Initial MATIC balance will be stuck in the contract after upgrade
Severity	**High**
Description	The current on-chain implementation has an idle amount of MATIC tokens, at the time of writing it is the following amount:  Matic Token (MATIC) $6,089.37 16,467.60328551 MATIC @0.3698 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 MATIC tokens.
Recommendations	Consider adjusting the stakeRewardsAndDistributeFees function to be compatible with MATIC tokens as well and consider compounding these idle rewards before the upgrade.
Comments / Resolution	Resolved.

Issue_08	Malicious user can DoS withdrawals by dusting the MaticX contract with small amounts of ValidatorShare tokens from different validators
Severity	**Medium**
Description	The requestWithdrawal function loops over all existing validators until either leftAmountToWithdraw = 0 or until all validators have been considered. In theory, this exposes an issue where the loop runs OOG at some point. Due to the fact that the architecture exposes a **preferred depositor**, 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. However, we still found a way to execute this exploit. That being said, there are a two prerequisites that are needed: a) The architecture must expose a large amount of different existing validators b) A withdrawal attempt must result in one validator being depleted which triggers the loop continuation to other validators Prerequisite a) is currently **NOT GIVEN** based on the on-chain implementation. **This means this issue can only happen once more validators are being added.** Once these prerequisites are given, a user can trivially buy shares from different validators, transfer them to the MaticX contract which will then result in amountToWithdrawFromValidator > 0 and attempts to loop over all validators which will potentially run OOG. PoC: 1. The registry contains a ton of validators, and most of them are not currently used by the MaticX contract.

	2. An attacker sees that and decides to stake a dust amount of POL in all empty validators. 3. The attacker directly transfers all those validators' shares to the MaticX contract. 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. 5. Because the function sellVoucher_newPOL is gas-intensive within each validator, the transaction will possibly run out of gas trying to withdraw the dust amounts of POL from a ton of validators. 6. The withdrawal attempt reverts.
Recommendations	Consider ensuring that only a reasonable amount of validators exist in the registry.
Comments / Resolution	Acknowledged, the client will ensure that the amount of validators stays reasonable. Furthermore, the BOT role is assumed to be trusted.

Issue_09	Lack of reward compounding before submit allows users to extract value from the protocol
Severity	**Medium**

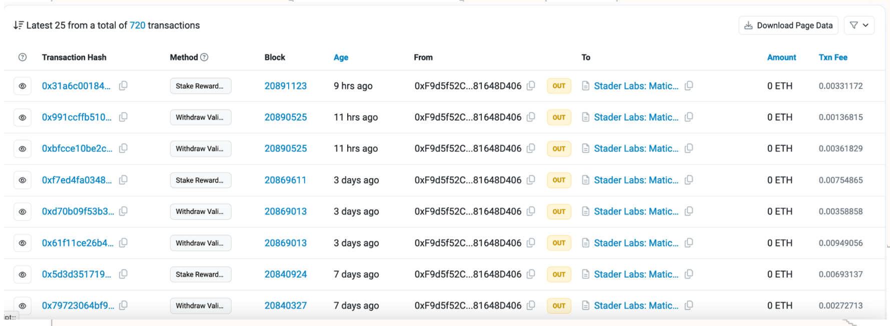
Description	Whenever users deposit or withdraw tokens, the current exchange rate via the _convertPOLToMaticX / _convertMaticXToPOL functions is used to determine how much MaticX will be received for staking POL tokens or how much POL tokens are received for redeeming MaticX.
	
	This exchange rate is dependent on the circulating MaticX supply and the total staked POL amount:
	
	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;

```
        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:

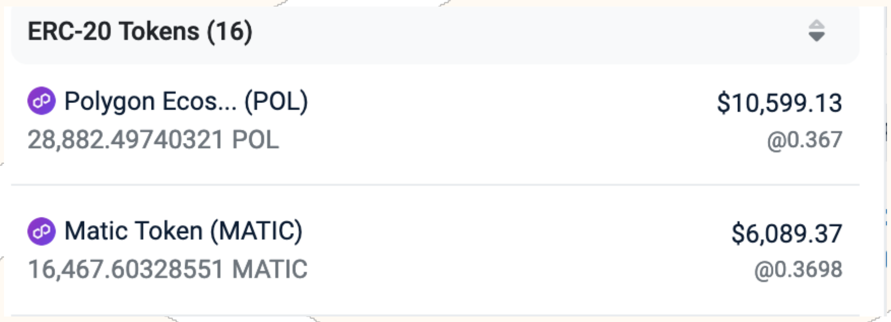


Latest 25 from a total of 720 transactions

Transaction Hash	Method	Block	Age	From		To	Amount	Txn Fee
0x31a6c00184...	Stake Reward...	20891123	9 hrs ago	0xF9d5f52C...81648D406	OUT	Stader Labs: Matic...	0 ETH	0.00331172
0x991ccffb510...	Withdraw Vali...	20890525	11 hrs ago	0xF9d5f52C...81648D406	OUT	Stader Labs: Matic...	0 ETH	0.00136815
0xbfcce10be2c...	Withdraw Vali...	20890525	11 hrs ago	0xF9d5f52C...81648D406	OUT	Stader Labs: Matic...	0 ETH	0.00361829
0xf7ed4fa0348...	Stake Reward...	20869611	3 days ago	0xF9d5f52C...81648D406	OUT	Stader Labs: Matic...	0 ETH	0.00754865
0xd70b09f53b3...	Withdraw Vali...	20869013	3 days ago	0xF9d5f52C...81648D406	OUT	Stader Labs: Matic...	0 ETH	0.00358858
0x61f11ce26b4...	Withdraw Vali...	20869013	3 days ago	0xF9d5f52C...81648D406	OUT	Stader Labs: Matic...	0 ETH	0.00949056
0x5d3d351719...	Stake Reward...	20840924	7 days ago	0xF9d5f52C...81648D406	OUT	Stader Labs: Matic...	0 ETH	0.00693137
0x79723064bf9...	Withdraw Vali...	20840327	7 days ago	0xF9d5f52C...81648D406	OUT	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) 28,882.49740321 POL	$10,599.13 @0.367
Matic Token (MATIC) 16,467.60328551 MATIC	$6,089.37 @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).

	6. The attacker profits 2,500 POL (52,500 withdrawn - 50,000 deposited). **Note**: 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.
Recommendations	Recommendations: Option 1: Consider invoking withdrawRewards (within a special iof-clause due to the minAmount requirement within ValidatorShare) and subsequently stakeRewardsAndDistributeFees (with the preferred deposit validator) to ensure that the exchangeRate is always up to date. Option 2: Consider incorporating the ValidatorShare's native restake function and trigger it on every deposit/redeem whenever the minAmount threshold is met. Option 3: Consider ensuring that the BOT invokes stakeRewardsAndDistributeFees regularly (every 3 hours as example), using tools like Chainlink Automation. We recommend going with Option 3 as this does not further modify the codebase (which prevents the introduction of undesired side-effects).
Comments / Resolution	Acknowledged, the client went with option 3.

Issue_10	feePercent change will be applied in hindsight
Severity	**Medium**
Description	The setFeePercent 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 stakeRewardsAndDistributeFees before any fee change. (If there any any idle rewards)
Comments / Resolution	Resolved, the _stakeRewardsAndDistributeFees function is invoked which will compound POL fees but not MATIC fees. This is however not a big issue since the contract only accumulates MATIC fees in edge-cases.

Issue_11	Cached requestEpoch during requestWithdraw will be inaccurate if WITHDRAWAL_DELAY is changed after a withdrawal has been requested
Severity	**Medium**
Description	A blunder within the requestWithdraw function will potentially disallow users to rightfully claim their withdrawal on time: *uint256 requestEpoch = stakeManager.epoch() + stakeManager.withdrawalDelay();* The requestEpoch is determined by using the current withdrawalDelay() at the time of requesting the withdrawal. This is incorrect due to the fact that the check within the ValidatorShare contract is as follows: *require(unbond.withdrawEpoch.add(stakeManager.withdrawalDelay()) <= stakeManager.epoch() && shares > 0, "Incomplete withdrawal period");* which is using the dynamic withdrawalDelay() value while the claimWithdrawal function uses requestEpoch which is corresponding to the withdrawalDelay at the time of the request creation. If the WITHDRAWAL_DELAY value is now decreased after a request has been made, users should theoretically be able to claim their request earlier (as per code within ValidatorShare). However, due to the blunder within the requestWithdraw function, this is impossible.
Recommendations	Consider following the same approach as the ValidatorShare contract by storing the currentEpoch into the WithdrawalRequest and applying the dynamic withdrawalDelay() on the check.

	Optionally, one can simply remove the epoch check within the MaticX contract as it would revert anyways within the ValidatorShare contract if the epoch for the nonce is not reached.
Comments / Resolution	Resolved.

Issue_12	Rare possibility of DoS'ing withdrawals by allocating dust reward amounts to many different validators in case of malicious BOT address
Severity	**Low**
Description	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. However, the root-cause of this issue is the fact that the BOT address can delegate funds to any validator via the stakeRewardsAndDistributeFees function (instead of only to the preferred depositor)
Recommendations	Consider strictly ensuring that: a) No unnecessary large amount of validators is listed within the ValidatorRegistry contract b) The BOT address is a (partially) trusted address
Comments / Resolution	Acknowledged, the client ensured that a) and b) will be enforced.

Issue_13	Integration Issue: Enforcement of POL instead of MATIC
Severity	**Low**
Description	Currently, the claimWithdrawal 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	Acknowledged.

Issue_14	Lack of reasonable upper limit for setFeePercent
Severity	**Low**
Description	The setFeePercent function allows setting the treasury fee of up to 100%. 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.
Recommendations	Consider changing this to a reasonable threshold (e.g. 10%)
Comments / Resolution	Resolved, this has been changed. Furthermore, the feePercent type was changed from uint8 to uint16. This should be carefully checked during the upgrade to ensure no collisions occur.

Issue_15	Trigger of temporary DoS of withdrawals due to validator unavailability by malicious actor
Severity	**Low**
Description	Within the *"Governance of ValidatorShare and StakeManager contracts has several privileges that can negatively impact MaticX"* 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 ValidatorShare or StakeManager contract. One explicit scenario is the scenario where the deactivationEpoch of a validator is above the current epoch: *else if (deactivationEpoch > currentEpoch) { // validator just unstaked, need to wait till next checkpoint* *revert("unstaking");* *}* which is the case (as the comment mentions) whenever a validator has just unstaked. In the scenario where there are no delegated stakes towards this validator, withdrawals will always work. However if a malicious user recognizes such a transaction by the validator and frontruns this with a dust purchase and transfer towards the MaticX contract, it may happen that this validator would then be part of the requestWithdraw loop which would then revert due to the above mentioned issue.
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	Acknowledged.

Issue_16	Reentrancy guard is only initialized during initializeV2
Severity	**Informational**

Description	The reentrancy guard is corresponding to values 1 and 2 for ENTERED and NOT_ENTERED:
	
	uint256 private constant NOT_ENTERED = 1;
	uint256 private constant ENTERED = 2;
	
	By default however, the value is zero:
	
	uint256 private reentrancyGuardStatus;
	
	which means that the very first function call will not be guarded with the reentrancy guard, if initializeV2 is not invoked beforehand. (after the first function call it is set to NOT_ENTERED)

Recommendations	Consider immediately calling initializeV2 after the proxy upgrade. Since this contract also functions without initializeV2, it is possible for users to interact with the contract immediately after the upgrade, before initializeV2 is called. (If the proxy upgrade and initializeV2 call are not in the same transaction). Optionally, one can mark it as NOT_ENTERED by default in the storage declaration.
Comments / Resolution	Acknowledged, the client ensured that initializeV2 will be called directly after the proxy upgrade.

Issue_17	Griefing: requestWithdraw before migrateDelegation can prevent migration
Severity	**Informational**
Description	Whenever the migrateDelegation function is called with the full existing balance of a validator , a user can simply invoke requestWithdraw with 1 wei beforehand which would then result in a revert of the migrateDelegation function because _amount 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 _amount parameter to match the balance.
Comments / Resolution	Resolved.

Issue_18	Griefing: Prevention of withdrawValidatorsReward
Severity	**Informational**
Description	The withdrawValidatorsReward 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 withdrawRewards function to withdraw rewards from one validator in the parameter list which then results in a revert of the withdrawValidatorsReward function call due to the following check within the ValidatorShare contract: *require(rewards >= minAmount, "Too small rewards amount");*
Recommendations	We do not see the necessity of a change. However, this should be kept in mind.
Comments / Resolution	Acknowledged.

Issue_19	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	Resolved. Furthermore, the feePercent type was changed from uint8 to uint16. This should be carefully checked during the upgrade to ensure no collisions occur.

Issue_20	_submit without preferred depositor being set will always result in using validatorId = 0
Severity	**Informational**
Description	The _submit function fetches the preferred depositor as follows: *uint256 preferredValidatorId = validatorRegistry .preferredDepositValidatorId();* *IValidatorShare validatorShare = IValidatorShare(stakeManager.getValidatorContract(preferredValidatorId));* If the preferred depositor is not set, this will always return zero, fetching the preferred validator with the ID = 0 (due to uint256 being by default 0). Fortunately, the validator with ID = 0 is not set within the StakeManager and always corresponds to address(0) which results in a revert.
Recommendations	We do not recommend a change. However, if still desired to fix this, consider simply reverting directly if the preferred depositor returns ID = 0.
Comments / Resolution	Acknowledged.

Issue_21	Setting DEFAULT_ADMIN_ROLE as roleAdmin for BOT role is redundant
Severity	**Informational**

Description	The DEFAULT_ADMIN_ROLE has by default all privileges to add/revoke roles due to the function returning 0x00 if no roleAdmin is set:
	
	`/**`
	`* @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;`
	`}`
	
	Therefore, it is not necessary to set it as role admin for the BOT role. If however, in the previous implementation a different roleAdmin has been set for the BOT role, this means that the DEFAULT_ADMIN_ROLE is no longer the roleAdmin and therefore a change is necessary to restore this state.

Recommendations	Consider thinking about if this change is necessary. Additionally we recommend adding tests to ensure the DEFAULT_ADMIN_ROLE is in fact the roleAdmin (which is the default case if nothing has been changed).
Comments / Resolution	Resolved, this change must be done because a different roleAdmin has been set in the previous iteration.

Issue_22	Arithmetic operations within ValidatorShare._buyShares can result in some dusted MATIC within MaticX contract
Severity	**Informational**
Description	The MaticX contract allows users to deposit POL and MATIC tokens. The ERC20 flow is as follows: a) Transfer MATIC from caller to MaticX contract b) Transfer MATIC from MaticX contract to StakeManager contract The arithmetic operations within ValidatorShare._buyShares are as follows: *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);* As one can already identify from the **comment**, it may be possible that _amount is less than the _amount parameter provided. This means if users want to stake MATIC instead of POL that eventually not all MATIC is being transferred to the StakeManager contract which results in some dust MATIC being locked within the MaticX contract. 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) However, given the existence of this comment, we are still of the opinion to raise this issue for eventual future upgrades.
Recommendations	Consider simply adjusting the stakeRewardAndDistributeFees function to also enable MATIC.

Comments / Resolution	Resolved.

Issue_23	Future Upgrade: Slashing after withdrawal request will result in incorrect output amount
Severity	**Informational**
Description	Currently, the StakeManager 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 withdrawExchangeRate in the corresponding ValidatorShare contract. When inspecting the overall business logic within the MaticX contract, users can always gauge how much POL tokens they will receive for the provided amount of MaticX using the following calculation: maticX * totalPol / totalMaticX This is also displayed within the convertMaticXToPol function. However, the Polygon staking architecture uses a slightly different calculation when transferring out POL tokens during the unstakeClaimTokens_newPOL function. A pool and share based approach is used which transfers out POL tokens based on the so-called withdrawExchangeRate(). 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) **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
Recommendations	We do not recommend a change as this is based on the underlying

	concept of the ValidatorShare contract which will also be influenced by requests outside from MaticX. A comment could be added which indicates that in such a scenario users may not receive the expected output amount.
Comments / Resolution	Acknowledged.

MaticX (New Deployment)

Disclaimer: This section of the report only displays issues which are related to a new deployment. Issues which are related to the existing deployment are displayed in the section above.

These observations represent extremely sophisticated edge cases and are very rare to be executed or take place.

Issue_24	Exotic edge-case will result in bricked _submit function
Severity	**APPLICABLE ONLY TO FORKING**
Description	A very exotic edge-case can result in a scenario where deposits are bricked and users will always lose all deposited POL/MATIC tokens. Consider a scenario where the MaticX contract is successfully operating for some time and at some point all users decide to redeem their MaticX for the underlying POL tokens. (This can be either all at once due to a specific event or step by step) An event could for example be the scenario if the usecase of MaticX is non-existent anymore (temporarily) because there are no yield sources where users can stake MaticX or rewards are paused for POL staking. However, unrelated to the reason for such a scenario, it is definitely possible for it to happen. Once the last user redeems their MaticX for POL tokens, that means the circulating MaticX supply becomes zero. But eventually there will be some idle rewards sitting in the contract. The BOT now automatically restakes these rewards and the contract is suddenly in the state where there are staked POL tokens without any circulating MaticX supply.

	If a new user deposits, the following calculation for POL -> MaticX is happening: *uint256 balanceInMaticX = (_balance * totalShares) / totalPooledAmount;* depending on the provided _balance and the totalPooledAmount (which was increased due to the restake), it can happen that the user will not receive any MaticX tokens and the provided POL tokens will be lost. PoC: - MaticX supply = 0 - totalPooledAmount = 1000e18 (due to BOT restaking) a) Alice deposits 100e18 POL tokens > (100e18 * 1) / 1000e18 = 0 b) Alice does not receive any MaticX tokens in exchange
Recommendations	Consider simply depositing a reasonable amount into the MaticX contract as governance or independent third-party which will never be withdrawn. This will ensure a scenario with zero circulating MaticX tokens and idle rewards can never happen.
Comments / Resolution	Acknowledged, the client will deposit a reasonable amount which will never be withdrawn.

Issue_25	MaticX conversion is susceptible to inflation attack (after new deployment)
Severity	**APPLICABLE ONLY TO FORKING**
Description	The MaticX contract is susceptible to the standard inflation attack which is widely known by an unconsidered edge-case. 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 POL/MATIC deposits, which will then result in the previous depositor receiving all / the majority of deposits. 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 MaticX as the exchange rate is not dependent on the ERC20 balance. Instead, there are two different scenarios of how this can be exploited: **First scenario (theoretical):** a) Depositing 1 wei of POL/MATIC (first depositor) b) Waiting until rewards are accrued and any address with the BOT role calls stakeRewardsAndDistributeFees c) The exchange rate is now successfully manipulated This scenario is rather theoretical than practical because the 1 WEI deposit will likely not accrue any real rewards **Second scenario (practical):** a) Depositing 1 wei of POL/MATIC (first depositor) b) Delegating funds from the own address to a ValidatorShare contract c) Transferring the ValidatorShare token directly to the MaticX contract d) The exchange rate is now manipulated

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("IValidatorShare", 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("1000000000000000001"));

	// 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);`
	
	`// 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"));`
	`});`
	
	Additionally, a malicious user can also frontrun the first deposit by purchasing ValidatorShare tokens and transferring them to the MaticX contract which breaks the ratio.

Recommendations	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. For future deployments, there are several options to prevent this: a) Include a minAmountOut parameter b) Transfer 1000 shares to 0xdead during the first deposit c) Prevent the scenario where zero shares are received d) Executing the first deposit after the deployment 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 MaticX tokens have been redeemed, there is still a small amount allocated to governance which prevents the init-state.
Comments / Resolution	Acknowledged, in future deployments it will be ensured that governance is the first depositor.