PUBLIC

Code Assessment

of the Vault Bridge Token Smart Contracts

May 8, 2025

Produced for

polygon

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Polygon Team,

Thank you for trusting us to help Polygon with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Vault Bridge Token according to Scope to support you in forming an opinion on their security risks.

Polygon implements an extension of the Unified Bridge (formerly LxLy Bridge) that enables the bridging of assets that have been deposited into an ERC-4626 yield-generating vault. Additionally, Polygon provides a Native Converter deployed on Layer Y that allows assets that were natively bridged (not via the VaultBridgeToken extension) to be converted to vault-bridged tokens, with the underlying token being bridged back to Layer X.

The most critical subjects covered in our audit are functional correctness, accounting correctness, and the integration with external systems (Bridge and Morpho vaults).

Functional correctness is good, after missing conversions between asset amounts and share amounts have been fixed, see drainVault Cannot Withdraw All Assets and Missing Asset-Share Conversions in Vaults. Accounting correctness is good, as related issues have been fixed, see drainVault Locks Assets. Security regarding integration with external systems is high.

In summary, we find that the codebase provides a good level of security.

The Notes section highlights behavior that users should be aware of.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	2
• Code Corrected	2
Medium-Severity Findings	3
• Code Corrected	3
Low-Severity Findings	3
• Code Corrected	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Vault Bridge Token repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	31 March 2025	634716469cc2d84346683ab3c39d45834581fa69	Initial Version
2	5 May 2025	c3b307ffd354002140f3db110ffebc0e6739bbf6	First Fixes Version
3	7 May 2025	8751f1a592470e95c4a78aa1c1808ba048eef581	Second Fixes Version

For the solidity smart contracts, the compiler version `0.8.28` was chosen.

As part of (Version 1), the following contracts are included in scope:

```
src/
    CustomToken.sol
    NativeConverter.sol
    TransferFeeUtils.sol
    VaultBridgeToken.sol
    VaultBridgeTokenInitializer.sol
    custom-tokens/
        GenericCustomToken.sol
        GenericNativeConverter.sol
        WETH/
            WETH.sol
            WETHNativeConverter.sol
    etc/
        ERC20PermitUser.sol
    vault-bridge-tokens/
        GenericVbToken.sol
        vbETH/
            VbETH.sol
        vbUSDT/
            TransferFeeUtilsVbUSDT.sol
```

Beginning with (Version 2), several contracts were renamed and additional ones were introduced. The updated scope includes everything from (Version 1), plus the following:

```
src/
    MigrationManager.sol
    ITransferFeeCalculator.sol
    vault-bridge-tokens/
        GenericVaultBridgeToken.sol (previously GenericVbToken.sol)
        vbUSDT/
            USDTTransferFeeCalculator.sol (previously TransferFeeUtilsVbUSDT.sol)
```

The compiler version used since (Version 2) is `0.8.29`.

The ERC4626 yield-generating vault configured in VaultBridgeToken was assumed to be a generic 4626 vault with only standard functionality, or a MetaMorpho V1.1 vault. This review did not consider any other underlying vaults.

2.1.1 Excluded from scope

Any other file not explicitly mentioned in the scope section is considered out of scope. In particular, the external systems such as the LxLy Bridge (Unified Bridge) and yield vaults (e.g. Morpho), the deployment scripts, the tests, and the configuration files are not part of this audit.

2.2 System Overview

This system overview describes the initially received version ((Version 1)) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Polygon offers an extension of the Unified Bridge (formerly LxLy Bridge) enabling the bridging of assets while depositing the accumulated liquidity into the ERC-4626 yield-generating vault. Additionally, Polygon provides a Native Converter deployed on Layer Y that allows assets that were natively bridged (not via the VaultBridgeToken extension) to be converted to assets bridged through the extension and bridged back to Layer X.

2.2.1 VaultBridgeToken

The VaultBridgeToken contract is an ERC-4626 vault. It enables bridging assets across supported chains while depositing the accumulated liquidity into another ERC-4626 yield-generating vault. The vault operates on a specified ERC-20 token, referred to as the *underlying token*, which is set during initialization. As yield is generated, it can be claimed by a designated yield recipient address. The contract maintains an internal reserve of underlying tokens, allowing users to withdraw from the vault without the need to redeem funds from the yield vault.

Users can deposit assets into the vault by calling the `deposit` function. In return, they receive the corresponding amount of vaultBridgeTokens (vbTokens), which represent shares according to the ERC-4626 standard. The conversion rate is fixed at 1:1, meaning the share price cannot change. If the deposited amount exceeds the `minimumYieldVaultDeposit` threshold, a portion of the assets may be forwarded to the yield-generating vault. Otherwise, the assets are held in the vault's internal reserve. The reserve is defined as a percentage of the total assets. If the existing reserve is below the target, newly deposited assets are used to replenish it until the target is met. Any remaining assets beyond the target are then deposited into the yield vault. As an alternative to the standard deposit, users can opt to call the `depositAndBridge` function. This mints vbTokens and bridges them directly to another network using the Unified Bridge. Users should ensure that they only bridge to supported network IDs. The vault also supports deposits via `depositWithPermit()` and `depositWithPermitAndBridge()`, which allow users to authorize deposits using an ERC-2612 permit signature instead of an approval. This allows depositing in a single transaction. Users can also choose to deposit by calling the ERC-4626 `mint` function, specifying the desired amount of vbTokens they want to receive.

Users can withdraw assets or redeem their vbTokens. Withdrawals or redemptions can be initiated by any address, as long as it has been granted sufficient allowance to spend the owner's vbTokens. The assets are first sourced from the vault's internal reserve. If the reserve does not contain enough to fulfill the request, the vault will attempt to withdraw the remaining assets from the yield-generating vault. If the combined total of the reserve and the yield vault is still insufficient, the transaction will revert. Additionally, when vbTokens are bridged from another network to the one where the vault contract resides, users can retrieve their assets in a single step by calling the `claimAndRedeem` function.

The vault accepts underlying assets migrated from designated NativeConverter contracts operating on other chains. These migrations happen through the Unified Bridge, which transfers both the underlying tokens and the corresponding cross-chain message. To successfully complete a migration, both `claimAsset` and `claimMessage` functions must be called on the Unified Bridge contract:

- `claimAsset` transfers the migrated underlying tokens to the vault.
- `claimMessage` triggers the `onMessageReceived` function on the vault, delivering the associated instructions.

When `onMessageReceived` is invoked, the vault decodes the message. If it's a `COMPLETE_MIGRATION` instruction, `_completeMigration()` is called to process the migrated tokens. The migrated tokens are deposited. The minted vbTokens are then bridged to the zero address on the chain where the migration originated. These vbTokens are not claimable, but they provide liquidity to the bridge, allowing users to bridge vbTokens that have been minted by a NativeConverter back to the source network.

If the actual internal reserve deviates from the target, a rebalancing operation can be triggered to correct the imbalance.

The vault uses role-based access control with four roles, all initially assigned to the same owner address.

- **Default Admin**
    - Set the `yieldRecipient` (after collecting pending yield).
    - Set `transferFeeUtil` for handling fee-on-transfer tokens.
    - Set `minimumReservePercentage`.
    - Set `nativeConverters`.
    - Set `minimumDepositAmount` for yield vault deposits.
    - Call `drainVault()` to withdraw from the yield vault.
    - Change the yield vault. (Before calling this, it should be ensured that all funds have been withdrawn from the old vault).
    - Unpause the vault.
    - Set the roles.
- **Pauser**
    - Can pause the vault.
- **Rebalancer**
    - Can rebalance the internal reserve.
- **Yield Collector**
    - Can collect generated yield, sending it to the `yieldRecipient`.

Two implementations of the VaultBridgeToken are provided:

- **GenericVaultBridgeToken:** The GenericVbToken is a basic implementation of VaultBridgeToken intended for general-purpose use.

- **VbETH:** The VbETH token is a specialized VaultBridgeToken with WETH as the underlying token. It allows users to deposit ETH directly via the `depositGasToken`, `depositGasTokenAndBridge`, and `mintWithGasToken` functions. VbETH manages wrapping ETH deposits into WETH. Users always receive WETH back from the contract. There is no function to withdraw and unwrap back to native ETH.

2.2.2 TransferFeeUtil

TransferFeeUtils is an abstract contract providing a standardized interface for calculating token transfer fees. It defines two primary methods: `assetsAfterTransferFee`, which calculates the amount of assets remaining after applying transfer fees, and `assetsBeforeTransferFee`, which determines the required assets before a transfer to achieve a desired post-transfer amount.

A specific implementation of TransferFeeUtils for USDT tokens is provided by TransferFeeUtilsVbUSDT. This implementation caches USDT's fee parameters (`cachedBasisPointsRate` and `cachedMaximumFee`) to efficiently compute transfer amounts according to USDT's fee structure. Additionally, it includes functionality (`recacheUsdtTransferFeeParameters()`), which the owner can call to pull these parameters directly from the USDT token contract.

2.2.3 CustomToken

CustomToken is an ERC-20 token designed to represent bridged vault tokens from Layer X on Layer Y. It supports ERC-20 and ERC-2612 permit functionality. Minting and burning operations are limited to the Unified Bridge and Native Converter contracts. Additionally, it provides a pausing mechanism controlled by the owner.

Two specific implementations of CustomToken are provided:

- GenericCustomToken offers an implementation suitable for general use cases without special requirements.
- WETH Custom Token is a specialized implementation representing VbETH. It includes `deposit` and `withdraw` functions for converting between ETH and WETH Custom Tokens. Furthermore, it provides a restricted `bridgeBackingToLayerX` function that enables the Native Converter to transfer ETH to itself and then back to Layer X. Note that this contract should only be used on a chain where the native token is ETH.

2.2.4 NativeConverter

The NativeConverter is deployed on Layer Y and can be used to convert underlying tokens (typically bridge-wrapped versions of Layer X tokens) into CustomTokens and vice versa. It also supports migrating backing assets to Layer X, where equivalent vbTokens are minted and bridged, which allows the vbTokens minted on Layer Y to be bridged back without deconverting first.

Users can convert underlying tokens into CustomTokens using `convert` or `convertWithPermit` and can deconvert using `deconvert`, `deconvertWithPermit`, `deconvertAndBridge`, and `deconvertWithPermitAndBridge`. The conversion rate is always 1:1.

NativeConverter includes a `migrateBackingToLayerX` function, which transfers underlying tokens to Layer X and sends a message to mint the vbTokens. This message will be received in the VaultBridgeToken's `onMessageReceived` function to complete the migration. The contract tracks the amount of underlying held as backing, (`backingOnLayerY`) and enforces a configurable threshold (`maxNonMigratableBackingPercentage`) to ensure sufficient liquidity is available for local redemptions.

Only the contract's `owner` can pause/unpause the contract and set the `migrator` role, as well as the `maxNonMigratableBackingPercentage`. The `migrator` is authorized to trigger migrations.

Two implementations of the Native Converter are provided:

- GenericNativeConverter is a direct extension of NativeConverter used for deployments where no special behavior is required.
- WETHNativeConverter is a specialized version used in combination with the WETH Custom Token. It includes a `migrateGasBackingToLayerX` function that enables migration of ETH backing to Layer X, using a custom cross-network instruction.

2.3 Roles and Trust Model

The `Default Admin` in the VaultBridgeToken is fully trusted. In the worst case, a compromised admin could set a malicious `yieldVault`, allowing them to steal all deposited funds. The `Pauser` role is partially trusted. If compromised, it can pause the vault, leading to a temporary denial of service. However, the default admin can unpause and revoke the role. The `Rebalancer` and `Yield Collector` roles are not trusted. The `yieldRecipient` is trusted to use any collected yield as expected.

The `owner` in the CustomToken contract is fully trusted. They can pause transfers of tokens, which could in the worst case lead to funds being frozen forever.

For all contracts that are deployed using upgradeable proxies, the proxy `owner` is fully trusted. In the worst case, a compromised owner can completely replace the contract with a malicious one, which could lead to the loss of all deposited user funds. The proxy contracts and configuration were out of scope for this review.

The Polygon LxLy Bridge (aka Unified Bridge) is fully trusted and assumed to always correctly bridge assets and messages. The trust model of the bridge applies. If the *emergency state* is activated in the zkEVM contract, the bridge is paused, which would temporarily disable bridging.

The Bridge Manager role in the BridgeL2SovereignChain contract is fully trusted. It can set custom mappings for any token. In the worst case, a compromised bridge manager could remap tokens to malicious contracts and mint infinite tokens, allowing them to steal all funds that have been bridged. The `owner` role of the upgradeable proxy of the PolygonZkEVMBridgeV2 contract is also fully trusted. In the worst case, it can upgrade the contract and steal all tokens that have been bridged.

The `yieldVault` is generally fully trusted. If there are any losses in the yield vault, the VaultBridgeToken can become partially unbacked, which would lead to a bank-run scenario where the last users would not be able to withdraw. However, tokens can be donated to the VaultBridgeToken in this situation to cover the loss, if someone is willing to do so.

The ERC20 tokens used as underlyingToken of the VaultBridgeToken and NativeConverter are expected to be standard tokens with no special functionality, aside from fees on transfer. In particular, tokens with transfer hooks (reentrancy), and rebasing tokens are not supported. Additionally, tokens with zero decimals are not supported.

If MetaMorpho V1.1 vaults are used as `yieldVault` in VaultBridgeToken, the following applies:

Morpho vaults contain trusted roles: The Owner is partially trusted. It can set all other roles and perform their actions. Additionally, it can set the fee to a maximum of `50%` of accrued yield at any time. However, it cannot charge a fee on the principal. It can increase the supply cap of a market or force the removal of a market. Both of these actions have a minimum 24-hour time-lock. If a risky/malicious market is added, this could lead to losses. If a market is removed, this can lead to the loss of the funds deposited in that market. In the worst case, users and/or the VaultBridgeToken `Default Admin` would have the time-lock (at least 24h) window to withdraw funds before the malicious action is performed. It may not be possible to withdraw everything depending on available liquidity in the yield vault. The Curator role is partially trusted and can also add/remove markets. However, the owner and the Guardian role (if it is set) can revoke any time-locked action. As a result, the trust required in the Owner and Curator is reduced if the Guardian and/or Owner can be trusted. Setting the Guardian role also requires waiting for a time-lock. This means the Guardian can revoke its own removal. In conclusion, as long as either the Owner or Guardian are behaving honestly, the worst misconfigurations can be avoided. If both cannot be trusted, malicious changes can only happen after the time-lock of at least 24 hours. The Guardian should be set and inspect any actions of the Owner and Curator carefully. The time-lock can be configured to be longer than 24h, which may be desirable. See Morpho Timelock Configuration.

While behaving honestly, the Curator should ensure that the markets added are considered within the desired risk parameters. The Allocator role is partially trusted. It can change the order in which assets are deposited/withdrawn to/from allowed markets, but it cannot add or remove a market. It should withdraw from and ensure that funds are not added to markets that have suffered losses or that were previously considered safe and are now considered unsafe. The Allocator should not add the same market to the `supplyQueue` multiple times.

The Morpho vaults must have an initial deposit from their deployer. See: Morpho Vault must Have Initial Deposit.

2.3.1 Changes in Version 2

The following changes were made in Version 2 of the contracts:

- The MigrationManager contract now handles the migration of funds from Layer Y to Layer X, a functionality previously integrated within VaultBridgeToken. This contract is designed to manage multiple VaultBridgeToken instances and stores a mapping that links Layer Y IDs and NativeConverters to TokenPairs. A TokenPair consists of a VaultBridgeToken and its corresponding underlyingToken. The admin of the MigrationManager contract can update this mapping via the `configureNativeConverters` function. The MigrationManager implements the `onMessageReceived` function. This function decodes messages sent by the NativeConverter on Layer Y and triggers the `completeMigration` function in the VaultBridgeToken contract, which transfers the migrated funds from the MigrationManager contract to the VaultBridgeToken. Each VaultBridgeToken managed by the MigrationManager contract has full allowance for the corresponding underlyingToken owned by the contract.
- The NativeConverter contract was updated to store the MigratorManager address, while the addresses for the corresponding VaultBridgeToken and the migrator role were removed. Access control changed from a single-owner model to a role-based model, introducing the Admin, Migrator and Pauser roles. The `deconvertWithPermit` and `deconvertWithPermitAndBridge` functions have been removed.
- The CustomToken contract moved from a single-owner model to a role-based model, introducing the Admin and Pauser roles.
- The WETHNativeConverter now maintains a minimum reserve of gas token, held within the WETH contract, in addition to the minimum reserve of wrapped ETH held in the NativeConverter. The effective minimum reserve is the combined total of both the gas token and wrapped ETH minimum reserves. Note that for an equal `nonMigratableBackingPercentage`, the WETHNativeConverter thus has a backing twice as large as that of a GenericNativeConverter.
- The TransferFeeUtils contract has been renamed to ITransferFeeCalculator and changed into an interface. The implementation of functions to compute assets amounts before and after transfer fees have been unified in the abstract VaultBridgeToken, streamlining the implementations of the concrete contracts GenericVbToken (now renamed GenericVaultBridgeToken) and VbETH.
- The TransferFeeUtilsVbUSDT contract has been renamed to USDTTransferFeeCalculator. The caching mechasim for fee parameters has been removed. Fee parameters are now retrieved directly from the USDT contract every time the fee is computed.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	2

- WETH and WETHNativeConverter Can Be Reinitialized Code Corrected
- drainVault Locks Assets Code Corrected

Medium-Severity Findings	3

- Incorrect Nonmigratable Backing Check in WETHNativeConverter Code Corrected
- Missing Asset-Share Conversions in Vaults Code Corrected
- drainVault Cannot Withdraw All Assets Code Corrected

Low-Severity Findings	3

- Missing Revert With Unsupported CustomCrossNetworkInstruction in VbETH Code Corrected
- No Nonmigratable Backing for WETH Code Corrected
- onMessageReceived May Be Called Before claimAssets Code Corrected

Informational Findings	7

- Incorrect Custom Token Storage Slot Code Corrected
- Initializer Can Be Called Directly Code Corrected
- Initializer Uses Outdated Naming Code Corrected
- Missing Events Code Corrected
- Small Deposits Into the Yield Vault Are Allowed Code Corrected
- Unused Function Parameters Code Corrected
- VaultBridgeToken Does Not Inherit IBridgeMessageReceiver Code Corrected

6.1 WETH and WETHNativeConverter Can Be Reinitialized

Security High Version 1 Code Corrected

CS-POL-VBT-001

The WETH and WETHNativeConverter contracts expose a `reinitialize` function. If only `initialize()` is called during deployment, the contracts remain unprotected against reinitialization. This allows any user to call `reinitialize`. In WETH, this enables setting a new NativeConverter and owner, granting mint/burn privileges. In WETHNativeConverter, reinitialization allows replacing the owner and underlying token, enabling arbitrary minting of WETH if a malicious mintable token is set.

The function is preceded by a comment "//@todo Remove."

Code corrected:

Both contracts have been modified to comment out the `reinitialize` function.

6.2 drainVault Locks Assets

Correctness **High** **Version 1** **Code Corrected**

CS-POL-VBT-002

The `drainVault` function withdraws assets from the `yieldVault` to the VaultBridgeToken but does not update `reservedAssets`, leaving the withdrawn assets untracked in the internal accounting. These assets cannot be withdrawn by users or redeposited in the `yieldVault`, creating a mismatch between issued shares and available assets. This accounting inconsistency would lead to insolvency of the contract, leading to a bank-run scenario if it is not corrected by admin action.

The accounting could only be corrected if the admins perform actions that should not be necessary for normal operation, such as upgrading the contract or intentionally setting an incorrect NativeConverter.

Code corrected:

In Version 2, the `drainVault` function was renamed to `drainYieldVault()`. The received amount is now correctly added to the `reservedAssets`.

However, the `shares` that are withdrawn from the `yieldVault` are no longer capped to the `maxRedeem` value. As a result, the `redeem()` call to the vault can now revert if the `shares` are greater than the `maxRedeem` value. This should only happen if the `exact` parameter is set to `true`.

In Version 3, when `exact` is false, the `shares` that are withdrawn from the `yieldVault` are correcty capped to the `maxRedeem` value.

6.3 Incorrect Nonmigratable Backing Check in WETHNativeConverter

Correctness **Medium** **Version 2** **Code Corrected**

CS-POL-VBT-019

In Version 2, a `nonMigratableBacking` limit was introduced for ETH that can be bridged back to Layer X in the `migrateGasBackingToLayerX()` of WETHNativeConverter. However, this limit relies on the `backingOnLayerY` variable, which is not increased when someone deposits ETH to the custom WETH contract, and is not decreased when ETH is migrated.

In particular, there can be situations where it is impossible to migrate anything back.

Consider the following situation: All custom WETH have been minted through the `deposit` function or bridged. None have been converted. This means `backingOnLayerY` is zero, but there can be a large amount of ETH in the contract. Now none of the ETH can be bridged back, as the `migratableAmount` is zero.

Code corrected:

In Version 3, a new function `migratableGasBacking` was added and is used for the check instead of `migratableBacking`:

```
function migratableGasBacking() public view returns (uint256) {
        uint256 nonMigratableGasBacking =
            _convertToAssets(Math.mulDiv(customToken().totalSupply(), nonMigratableBackingPercentage(), 1e18));

        uint256 gasBalance = address(customToken()).balance;

        return gasBalance > nonMigratableGasBacking ? gasBalance - nonMigratableGasBacking : 0;
    }
```

```
uint256 migratableBacking_ = migratableBacking();

    // ...

require(amount <= migratableGasBacking(), AssetsTooLarge(migratableBacking_, amount));
```

With this change, the contract always preserves at least

```
_convertToAssets(customToken().totalSupply() * nonMigratableBackingPercentage / 1e18)
```

wei of ETH. Note that the backing as ERC20 (`$.backingOnLayerY`) must independently stay above the same value.

6.4 Missing Asset-Share Conversions in Vaults

Correctness **Medium** **Version 1** **Code Corrected**

CS-POL-VBT-003

There are missing conversions between asset amounts and share amounts in the VaultBridgeToken and NativeConverter contracts.

In VaultBridgeToken:

1. In the `_calculateAmountToReserve` function, the `minimumReserve` variable is computed by applying a percentage to the total number of shares (`totalSupply() + nonMintedShares`). This results in a value expressed in shares. However, this value is later compared to an amount in assets (`$.reservedAssets`), without converting the shares to their equivalent asset amount.
2. The `yield` function returns the amount of assets generated as yield. However, in `collectYield()`, this asset amount is passed to the `_mint` function, which expects an amount in shares.
3. In the `collectYield` function, the `$.netCollectedYield` variable is updated with the asset amount returned by `yield()`. The `_burn` function, however, updates `$.netCollectedYield` with a share amount.

In NativeConverter:

4. In the `migrateBackingToLayerX` function, the `maxNonMigratableBacking` value is computed by applying a percentage to `totalToken`, which represents a custom token amount (analogous to shares in standard ERC4626). This value (`maxNonMigratableBacking`) is then compared to `$.backingOnLayerY`, which represents an underlying token amount (analogous to assets in standard ERC4626).

In the current version, the asset-to-share ratio is hard-coded to always be 1-to-1. As a result, the missing conversions do not currently lead to incorrect behavior.

However, the mismatch would lead to incorrect accounting if the asset-to-share ratio can change in a future version of the code. Other parts of the code do handle the case where the asset-to-share ratio can change.

Code corrected:

VaultBridgeToken has been modified to include the following changes:

1. In the `_calculateAmountToReserve` function, the `minimumReserve` variable is now converted to its equivalent asset amount using the `convertToAssets` function before being compared to `$.reservedAssets`.

2-3. The `yield` function now returns the amount of shares generated as yield.

4. The code has been updated to use a new function (`migratableBacking()`) which computes the migratable backing in assets directly.

6.5 drainVault Cannot Withdraw All Assets

Correctness Medium Version 1 Code Corrected

CS-POL-VBT-005

There are missing conversions between asset amounts and share amounts in `drainVault()` in the VaultBridgeToken contract, which reduce the amount that can be withdrawn:

- The `totalAmount` variable represents the number of yield vault shares held by VaultBridgeToken. `maxAmount` indicates the maximum quantity of assets that VaultBridgeToken can withdraw from the yield vault. These two values are compared to determine the actual amount that can be withdrawn. This comparison is incorrect, as we are comparing a share amount to an asset amount. If the yield vault share price is greater than one, we will withdraw fewer assets than intended. It will not be possible to fully drain the vault in a single call to `drainVault`. Note that the `totalAmount` limitation is unnecessary even if implemented correctly, as the `maxWithdraw` function in ERC4626 is already capped to the balance of the caller.
- The `result` variable represents a number of shares, which is compared to `amountToDrain_`, an asset amount. If the `exact` argument is set to true, the transaction is highly likely to fail.

Example for the first case (assuming a 1:2 share-to-assets rate):

```
// Initial values
amountToDrain_ == 4     // Assets to withdraw from the yield vault
totalAmount == 2        // Yield vault shares owned by VaultBridgeToken
maxAmount == 4          // Maximum assets that can be withdrawn from the yield vault

// After comparison
maxAmount == 2          // Capped by totalAmount
amountToDrain_ == 2     // Capped by maxAmount
```

As a result, 2 assets will remain in the vault after the withdrawal, even though all assets could have been withdrawn.

Code corrected:

In Version 2 the `drainVault` function was rewritten and renamed to `drainYieldVault()`. It now correctly handles the asset and share amounts.

6.6 Missing Revert With Unsupported CustomCrossNetworkInstruction in VbETH

Correctness Low Version 1 Code Corrected

CS-POL-VBT-006

When the VaultBridgeToken contract receives a message from the Unified Bridge, it processes it via the `onMessageReceived` function. The function decodes the message into a `CrossNetworkInstruction`. If the instruction is of type `CUSTOM`, it delegates the handling to the `_dispatchCustomCrossNetworkInstruction` function. As indicated in the contract's comments, `_dispatchCustomCrossNetworkInstruction()` must revert if it encounters an unsupported instruction.

The VbETH contract inherits from VaultBridgeToken and overrides `_dispatchCustomCrossNetworkInstruction`. However, it does not revert when given an unsupported instruction. As a result, the message is consumed from the Unified Bridge but effectively ignored by VbETH.

Code corrected:

In Version 2, the handling of `CrossNetworkInstruction` has been updated. The `onMessageReceived` function has been moved to the new MigrationManager contract, which centralizes migration handling across all VaultBridgeToken instances. The `CUSTOM` instructions are no longer supported, eliminating the need for reverts.

6.7 No Nonmigratable Backing for WETH

Design | Low | Version 1 | Code Corrected

CS-POL-VBT-007

In NativeConverter, there is a configurable `maxNonMigratableBackingPercentage`, which sets a minimum amount that cannot be migrated back to Layer X. This is used to allow some liquidity for `deconvert` calls.

The WETHnativeConverter does not have such a nonmigratable backing configuration for native ETH tokens migrated in the `migrateGasBackingToLayerX` function. As a result, a migration could lead to a situation where a migration can drain all ETH from the WETH contract, which would not allow `withdraw` to be called to receive ETH.

Code corrected:

In Version 2, the following restriction was added to the `migrateGasBackingToLayerX` function:

```
uint256 migratableBacking_ = migratableBacking();

// ...

require(amount <= migratableBacking_, AssetsTooLarge(migratableBacking_, amount));
```

However, this introduced a new issue. See Incorrect nonmigratable backing check in WETHNativeConverter.

6.8 onMessageReceived May Be Called Before claimAssets

Design | Low | Version 1 | Code Corrected

In VaultBridgeToken, the migration flow should be as follows:

1. The `migrator` calls `migrateBackingToLayerX()` on Layer Y.
2. Someone calls `claimAssets()` in the bridge contract on Layer X.
3. Someone calls `claimMessage()` in the bridge contract on Layer X, which triggers `onMessageReceived()` in VaultBridgeToken, to finish the migration.

However, the order of 2. and 3. is not enforced. As a result, the `_completeMigration` function may increase the `reservedAssets` and mints new shares even though the migrated assets are not in the contract yet. This temporarily leads to incorrect accounting. However, the `claimAssets` function can eventually be called by anyone, at which point the accounting will be correct again.

An unlikely scenario where the accounting could be incorrect for a longer period of time is if the `onMessageReceived()` call is successful, but then the emergency state is triggered in the bridge. In this case, a subsequent call to `claimAssets()` will only be possible once the emergency state is over.

Code corrected:

In Version 2, the MigrationManager contract is introduced, which is now used to receive funds and call `completeMigration()`. `completeMigration()` now transfers the migrated funds from the MigrationManager to the VaultBridgeToken. This enforces that the assets must be claimed first, as otherwise the transfer would revert.

6.9 Incorrect Custom Token Storage Slot

Informational Version 1 Code Corrected

CS-POL-VBT-011

The CustomToken contract sets the `_CUSTOM_TOKEN_STORAGE` constant to `0x5bbe451cf8915ac9b43b69d5987da5a42549d90a2c7cab500dae45ea6889c900`, which does not match the ERC-7201 storage slot derived from the string `0xpolygon.storage.CustomToken`.

The code comment correctly states that the value should be computed as:

```
keccak256(abi.encode(uint256(keccak256("0xpolygon.storage.CustomToken")) - 1)) & ~bytes32(uint256(0xff))
```

However, the value used is not the result of this calculation.

Code corrected:

The constant has been updated to the correct value matching the ERC-7201 storage slot derived from the string `0xpolygon.storage.CustomToken`.

6.10 Initializer Can Be Called Directly

Informational Version 1 Code Corrected

CS-POL-VBT-012

The VaultBridgeTokenInitializer contract is designed to be called only through a delegatecall in the initializer function of the VaultBridgeToken contract. However, it can be called directly, which is not intended. The `__VaultBackedTokenInit_init` function could be disabled when not called from the initializer.

Code corrected:

In Version 2, `__VaultBackedTokenInit_init()` has been renamed to `initialize()`. The function now includes the `onlyDelegateCall` modifier, ensuring it can only be invoked via delegatecall. However, it could still be called outside the initialization flow of the callee contract.

6.11 Initializer Uses Outdated Naming

Informational Version 1 Code Corrected

CS-POL-VBT-013

The VaultBridgeTokenInitializer has the function `__VaultBackedTokenInit_init`, which uses the outdated naming convention "backed token" instead of "bridged token".

Code corrected:

In Version 2, the function was renamed to `initialize()`.

6.12 Missing Events

Informational Version 1 Code Corrected

CS-POL-VBT-014

There are missing events in the VaultBridgeToken and NativeConverter contracts.

- In VaultBridgeToken, the `changeYieldVault`, `setMinimumDepositAmount`, and `drainVault` functions do not emit any events.
- In NativeConverter, the `setMigrator` function does not emit an event.

Other similar functions in the codebase do emit events.

Code corrected:

In Version 2, the `setMigrator` function has been removed from NativeConverter. In VaultBridgeToken, `changeYieldVault()` has been renamed to `setYieldVault()`, and `drainVault()` to `drainYieldVault()`, both of which now emit their respective events. Polygon decided that `setMinimumDepositAmount()` should not emit an event.

6.13 Small Deposits Into the Yield Vault Are Allowed

Informational Version 1 Code Corrected

CS-POL-VBT-015

In VaultBridgeToken, the `minimumYieldVaultDeposit` variable defines the threshold for initiating a deposit into the yield-generating vault. However, when `assetsToReserve` is close to `assets`, an amount smaller than this can be deposited. This is because a part of the amount will go to the reserve.

Code corrected:

In Version 2, the code has been updated to first compute the assets to deposit and then check if the amount is bigger than the minimum yield vault deposit. This way, the amount deposited into the yield vault will always be bigger than `minimumYieldVaultDeposit`

6.14 Unused Function Parameters

Informational Version 1 Code Corrected

CS-POL-VBT-017

In VaultBridgeToken, the `_withdrawFromYieldVault` function has two unused parameters, `originalTotalSupply` and `originalYield`. They are only used in the commented-out code, which is marked with "TODO: find a way to test this check".

Code corrected:

In Version 2, `originalYield` has been renamed to `originalUncollectedYield` and the previously commented-out checked has been modified and added back to the code. All parameters are now used in the function.

6.15 VaultBridgeToken Does Not Inherit IBridgeMessageReceiver

Informational Version 1 Code Corrected

CS-POL-VBT-018

The docs of the LxLy bridge suggest that message receivers should implement the IBridgeMessageReceiver interface. While the VaultBridgeToken does implement the function `onMessageReceived`, it does not inherit the IBridgeMessageReceiver interface.

Code corrected:

In Version 2, the MigrationManager contract is the message receiver and implements the IBridgeMessageReceiver interface.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Inconsistencies in vbETH

Informational Version 1 Acknowledged

CS-POL-VBT-010

In VbETH, the `depositGasTokenAndBridge` function has a parameter named `destinationAddress`. In other functions, the equivalent parameter is named `receiver`.

Additionally, the `depositGasTokenAndBridge` function does not check if the provided `destinationNetworkId` is the contract's network ID. The `depositAndBridge` function in VaultBridgeToken reverts in this case.

Acknowledged:

Polygon chose to revise the contract later, stating: "Will be revisited later".

7.2 Open TODOs

Informational Version 1 Code Partially Corrected

CS-POL-VBT-004

As of Version 1 the following open TODOs are present in the codebase:

- In VaultBridgeToken:
  - Verify there cannot be reentrancy in the `_deposit`, `_rebalanceReserve`, `_completeMigration`, and `_withdrawFromYieldVault` functions.
  - Review hardcoded values in `claimAndRedeem()`.
  - Use the old optimization in `claimAndRedeem()`.
  - Reintroduce the check in `_withdrawFromYieldVault()`.
- In TransferFeeUtilsVbUSDT: Document `assetsAfterTransferFee()` and `assetsBeforeTransferFee()`.
- In WETH: Remove `reinitialize()` and "make upgradable to enable future ETH staking plans".
- In WETHNativeConverter: Remove `reinitialize()` and SafeERC20 library.

The TODOs should be resolved before the code is deployed.

Code partially corrected:

The TODOs have been partially addressed. However, additional TODOs were added in Version 2 of the codebase, related to the compatability with testnet deployments. Some parts of the code have been commented out. Before the code is deployed, all TODOs should be resolved and the commented out code should be removed.

7.3 Unreachable Code in TransferFeeUtils

Informational **Version 1** **Acknowledged**

CS-POL-VBT-016

In USDTTransferFeeUtils, there is the following code:

```
uint256 feeCandidate = (candidate * cachedBasisPointsRate) / 10000;
    if (feeCandidate > cachedMaximumFee) {
        return minimumAssetsAfterTransferFee_ + cachedMaximumFee;
    }

    while (candidate > 0) {
        uint256 feeCandidateMinus1 = ((candidate - 1) * cachedBasisPointsRate) / 10000;
        if (feeCandidateMinus1 > cachedMaximumFee) {// unreachable if we didn't return above
            feeCandidateMinus1 = cachedMaximumFee;
        }
```

Here, the `if` condition in the `while` loop is unreachable, as feeCandidateMinus1 will always be less than `feeCandidate`. As a result, the code would already return in the first `if`, whenever the second condition is true.

Acknowledged:

Polygon has renamed the contract to USDTTransferFeeCalculator but chose to revise the contract later, stating "The contract will not be used in production at the moment (will be revisited later)".

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Bridging Back May Be Temporarily Unavailable

Note **Version 1**

When a user bridges back VaultBridgeToken from Layer Y to Layer X, they can usually claim their tokens immediately. However, there can be an edge case where there is not enough liquidity on Layer X to cover the claim. This can happen if there have been tokens minted through the nativeConverter on Layer Y, and the underlying assets have not been migrated back to layer X. If only such tokens are remaining on Layer Y, the user must wait until the `migrator` calls the `migrateBackingToLayerX` function to migrate the assets back to Layer X. This is unlikely to happen, as it requires that all tokens minted on Layer X have already been bridged back to Layer X.

If the user notices that there is insufficient liquidity, they can use the `deconvert` function of the nativeConverter to convert their tokens back to the underlying asset. Then they can bridge back the underlying asset to Layer X.

As a result, users should be aware that bridging back may be temporarily unavailable if there is not enough liquidity on Layer X to cover the claim. If there are many exits and the user is bridging large amounts, they may want to check for sufficient liquidity before bridging back.

8.2 Custom WETH Behavior

Note **Version 1**

The custom-mapped token for the vbETH vault will be called WETH. However, this token should not be expected to behave equivalently to the standard WETH9 contract, which is commonly referred to as WETH. It differs in the following ways:

- The custom WETH will not be fully redeemable by using the `withdraw` function, as some of the backing ETH will be on a different chain (in the vbETH vault). If there is no remaining ETH in the contract, users will need to bridge to the X chain in order to withdraw their ETH.
- The ETH backing the custom WETH will be used in the vbETH vault, where it can be deposited to an `underlyingVault`, which exposes it to the risks of that vault. This means that the custom WETH is not a simple 1:1 representation of ETH, as it is in the standard WETH9 contract. The custom WETH will be redeemable for ETH, but only to the extent that the vbETH vault has sufficient backing ETH available and immediately available on the X chain.
- The custom WETH can be turned into standard WETH using the `deconvert` function of the WETHnativeConverter, but only as long as there are enough standard WETH tokens held by the native converter.

Users should be aware of these differences when deciding to use or integrate with the custom WETH. If they are handling large amounts, they may need to bridge to Layer X to withdraw their ETH.

8.3 Morpho Timelock Configuration

Note **Version 1**

The `setMinimumReservePercentage` (with a subsequent `rebalanceReserve()` call) and `drainVault` functions can be used by the `DEFAULT_ADMIN_ROLE` to withdraw funds from the `yieldVault`. If the `DEFAULT_ADMIN_ROLE` wants to reduce the trust in the Morpho vault owner/guardian not to add/remove a bad market, (see Roles and Trust Model) the admin can withdraw all available funds (note that depending on liquidity only a partial withdrawal may be possible) from the vault, before the timelock of a market change expires. However, this is only possible if the timelock on the Morpho vault is set to a longer time than the timelock of the `DEFAULT_ADMIN_ROLE`.

To minimize trust assumptions, it may therefore be desirable to configure the timelock of the Morpho vault to a longer time than the timelock on actions of the `DEFAULT_ADMIN_ROLE`.

8.4 Morpho Vault Must Have Initial Deposit

Note **Version 1**

If the `yieldVault` of VaultBridgeToken is a Morpho vault, the vault must be initialized carefully to avoid a share inflation attack. The Morpho code comments state the following:

```
@notice For tokens with 18 decimals, the protection against the inflation front-running attack
is low. To protect against this attack, vault deployers should make an initial deposit of a
non-trivial amount in the vault or depositors should check that the share price does not exceed
a certain limit.
```

As a result, the vault deployer should make an initial deposit of a non-trivial amount. Otherwise, the vault should not be set as `yieldVault` of VaultBridgeToken. Once there are significant funds deposited, it is no longer possible to trigger the share inflation attack.

8.5 Withdrawing From VaultBridgeToken May Be Temporarily Unavailable

Note **Version 1**

When a user withdraws from VaultBridgeToken, they can usually claim their tokens immediately. However, there may be limited liquidity in the underlying vault (`yieldVault`) to service withdrawals. As a result, large withdrawals may not always be possible. If the `yieldVault` is a MetaMorpho vault, the deposited funds may be lent out to other users. If all funds are lent out, the user must wait until there are funds available again to withdraw. As the utilization would be very high in this case, the interest is expected to be very high. This incentivizes borrowers to pay back their loans quickly and also incentivizes other users to deposit their assets into the Morpho market.

Users should be aware that there can be liquidity constraints on withdrawing from the VaultBridgeToken. These constraints will likely only be relevant if there is a very large amount withdrawn in a short time.

8.6 totalAssets and totalSupply Do Not Count All Tokens

Note **Version 1**

The `totalAssets` and `totalSupply` functions only count the assets and supply for which the backing is present on the chain on which the VaultBridgeToken is deployed. However, there can also be tokens minted on other chains through the native converters. These will only be counted in the `totalAssets` and `totalSupply` functions once the `migrateBackingToLayerX` function has been called and the message has been claimed. If a `maxNonMigratableBackingAmount` is set, some tokens may never be counted in the `totalAssets` and `totalSupply` functions.

As a result, the `totalAssets` and `totalSupply` functions should not be relied upon by integrators and their results can suddenly increase when the `migrateBackingToLayerX` function is called.