PUBLIC

Code Assessment

of the Circle xReserve Smart Contracts

Feb 09, 2026

Produced for

by

Contents

1 Executive Summary

Dear Circle team,

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

The assessed contracts are part of Circle xReserve, a bridge that allows users to bridge assets between chains. Chains without support for native USDC tokens will deploy a USDC-backed stablecoin and the backing USDC will be held in the xReserve contract on Ethereum.

Only the smart contracts of the xReserve are assessed. We could not assess if the smart contract provide all necessary checks as some of the checks are performed off-chain. Circle did not provide sufficient documentation on the integration of the contracts into the off-chain system, nor was a list of guarantees provided that the on-chain part should cover. The bridge's off-chain system is vital for the system's security, and it must be assumed that Circle performs all checks not covered on-chain correctly off-chain. We provided a non-exhaustive list of checks that should be performed off-chain in On-chain and off-chain checks. Otherwise, the smart contracts are well-structured and implemented. Almost all issues are informational.

In summary, we find that the isolated smart contract provides a high level of security.

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	0
Medium-Severity Findings	0
Low-Severity Findings	2
• Code Corrected	1
• Risk Accepted	1

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 Circle xReserve 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	04 Sep 2025	0b5a49f95f19eac7ba81baf56abd903bab66a298	Initial Version
2	24 Sep 2025	3532478636c886656645f9b83ae3c1b2ac39ddb1	Fixed Version
3	03 Dec 2025	7b06de56aaaede8bd7aa060257d816ca4f7312d9	Public Version
4	23 Dec 2025	b2579dcbffa33f9b395ab61389f08f4df38015d4	Removal of Permit and Authorization
5	06 Feb 2026	a571cbe12fa7cede3dfd48bc4fedb74739c04377	Public Version

For the solidity smart contracts, the compiler version `0.8.29` was chosen. The following files were in the scope of this review:

```
common/
    Constants.sol
    Errors.sol
interfaces/
    IERC7597.sol
    IERC7598.sol
    IGatewayMinter.sol
    IGatewayWallet.sol
    IRemoteDomainDepositor.sol
    IRemoteDomainHookExecutor.sol
lib/
    AddressLib.sol
    DepositIntent.sol
    DepositIntentLib.sol
    DepositParams.sol
    NoValidationAttestationLib.sol
    WithdrawHookData.sol
    WithdrawHookDataLib.sol
modules/remote-domain-depositor/
    Attestable.sol
    DomainManageable.sol
modules/x-reserve
    Blocklistable.sol
```

```
    DepositToRemote.sol
    Domain.sol
    Immutables.sol
    Pausing.sol
    RemoteDomainRegistration.sol
    TokenSupport.sol
    Withdrawal.sol
RemoteDomainDepositor.sol
xReserve.sol (UnifiedBridge.sol in |ver1|)
UpgradeablePlaceholder.sol
```

2.1.1 Excluded from scope

All third-party code like libraries is excluded from the scope. The system relies heavily on off-chain operators controlled by Circle, as well as other building blocks previously implemented by Circle (Gateway and CCTP) which are not in scope of this review. Additionally, chain partners' on-chain operations on the remote chain and their off-chain operations are out of scope.

As a cross-chain primitive, the same code will be deployed on different blockchains. These chains differ in their behavior, and their behavior might change in the future. The assessment was done on Ethereum's behavior and EVM specifications. Before deploying on another chain, the compatibility needs to be assessed and tested thoroughly.

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.

Circle implements xReserve, a deposit-and-mint bridge that stores native USDC in a contract on the native chain (Ethereum) and allows Circle's partners to mint USDC-backed stablecoin on remote chains. Each USDC-backed stablecoin is fully backed by the deposited USDC on the source chain.

There are four roles in the system. These are not to be mixed with the roles described in the Trust Model section:

- **Circle**: Operates the source chain contract and manages the logic for holding USDC deposited under each chain partner's address.
- **Chain partners**: Implement and issue USDC-backed stablecoin on their respective destination domains, using the chain's native languages and token standards.
- **End users**: Deposit USDC as collateral on the source chain (Ethereum) and interact with USDC-backed stablecoin on the destination chain, relying on Circle's and the chain partner's correct implementation.

The xReserve system incorporates two other bridges implemented by Circle: the Gateway Protocol and the Cross-Chain Transfer Protocol (CCTP). There are two main smart contracts in the system, both of which inherit from several utility contracts:

- `xReserve`: Manages transfers of assets between different domains.
- `RemoteDomainDepositor`: Manages deposits to remote domains. Its address is used by the Gateway Protocol to track bridged assets and to mint/burn tokens.

Other relevant contracts are `GatewayWallet` and `GatewayMinter` from the Gateway Protocol, and `TokenMessenger` from CCTP. Note that two versions of CCTP exist; in the second version, the contract is called `TokenMessengerV2`. This detail is omitted in the remainder of the report.

2.2.1 Partner Registration

All USDC collateral deposited by xReserve end users is stored inside `GatewayWallet` under a per-domain `RemoteDomainDepositor` contract address created by `xReserve` at registration time. Funds leave `GatewayWallet` only when Circle issues a Gateway attestation.

Circle registers a chain by calling `xReserve.registerRemoteDomain()`. This deploys a new `RemoteDomainDepositor` contract for the chain and returns its address. All native USDC collateral for that chain is stored under the `RemoteDomainDepositor` address inside `GatewayWallet`.

All four asset flows are described in the following subsections. Chains A and B refer to domains with native USDC, while chains X and Y are operated by Circle's partners and support only the USDC-backed stablecoin.

2.2.2 From Chain A to Chain X

The flow for moving funds from a native USDC domain (Chain A) to xReserve remote domain (Chain X) is as follows:

1. A user calls `xReserve.depositToRemote()`.
    - `xReserve` deposits the funds into `GatewayWallet` under the `RemoteDomainDepositor` address for Chain X.
    - `xReserve` emits a `DepositedToRemote` event, which is detected by Circle's backend.
2. Circle's service updates the off-chain balance of the Chain X depositor and issues a Deposit attestation.
3. The Chain X operator retrieves this attestation from the xReserve API.
4. Using the attestation, the operator mints USDC-backed stablecoin on Chain X.

2.2.3 From Chain X to Chain A

The flow for redeeming USDC-backed stablecoin on Chain X and receiving native USDC on Chain A is as follows:

1. An end user burns USDC-backed stablecoin on Chain X. This triggers the Chain X operator to generate a signed `BurnIntent` for the Gateway Protocol.
2. The Chain X operator submits the `BurnIntent` to Circle's backend.
3. The backend service verifies the `BurnIntent`, decrements the Chain X's off-chain balance, and issues a signed Gateway attestation.
4. The Chain X operator uses the attestation on `GatewayMinter` on Chain A, releasing USDC to the end user.
5. Circle's backend service observes the `GatewayMint` event and burns the corresponding amount in `GatewayWallet` on Chain A.

Note that, Circle initially plans to deploy xReserve only on Ethereum. In the future, bridging to another Chain B will be possible as long as `GatewayMinter` is deployed there, even if it is not the chain where the original deposit occurred. This will not require changes at the smart contract level.

2.2.4 From Chain X to Chain B (via CCTP)

The flow for redeeming USDC-backed stablecoin on Chain X and receiving native USDC on Chain B goes through Chain A, which has `GatewayMinter` support, and uses CCTP for the final transfer. Chain B does not have `GatewayMinter` deployed. The steps are as follows:

1. An end user burns USDC-backed stablecoin on Chain X. This triggers the Chain X operator to generate a signed `BurnIntent` which contains a `TransferSpec`:
   - `TransferSpec.destinationRecipient` is set to the `xReserve` contract address on Chain A.
   - `TransferSpec.destinationCaller` is also set to the `xReserve` contract address on Chain A.
   - `TransferSpec.hookData` is in the format of `WithdrawHookData`, which contains the `CCTP.depositForBurn` calldata in the `forwardingCalldata` field.
2. The Chain X operator submits the `BurnIntent` to Circle's backend.
3. The backend verifies the `BurnIntent`, decrements the Chain X off-chain balance, and issues a signed attestation for forwarding to Chain A (where Gateway exists).
4. The attestation is relayed on Chain A through `xReserve.withdraw()`. The following steps happen automatically:
   - Native USDC is minted through `GatewayMinter` to the `xReserve` address.
   - `CCTP.depositForBurn()` is invoked to burn and bridge the funds to Chain B.
5. Circle's backend service observes the `GatewayMint` event and burns the corresponding funds held for the Chain X depositor address in `GatewayWallet`.

2.2.5 From Chain X to Chain Y (Redemption and Redeposit)

The flow for redeeming USDC-backed stablecoin on Chain X and receiving USDC-backed stablecoin on Chain Y goes through Chain A, which has `GatewayMinter` support. The steps are as follows:

1. An end user burns USDC-backed stablecoin on Chain X. This triggers the Chain X operator to generate a signed `BurnIntent` which contains a `TransferSpec`:
   - `TransferSpec.destinationRecipient` is set to the `xReserve` contract address on Chain A.
   - `TransferSpec.destinationCaller` is also set to the `xReserve` contract address on Chain A.
   - `TransferSpec.hookData` is in the format of `WithdrawHookData`, which contains `xReserve.depositToRemote` in the `forwardingCalldata` field.
2. The Chain X operator submits the `BurnIntent` to Circle's backend.
3. The backend verifies the `BurnIntent`, decrements the Chain X off-chain balance, and issues a signed Gateway attestation for forwarding to Chain A (where Gateway exists).
4. The attestation is relayed on Chain A through `xReserve.withdraw()`. The following steps happen automatically:
   - Native USDC is minted through `GatewayMinter` to the `xReserve` address.
   - `xReserve.depositToRemote()` is invoked: funds are deposited and a `DepositedToRemote` event for Chain Y is emitted.
5. Circle's backend detects the `DepositedToRemote` event, issues a Deposit attestation for Chain Y, and increments Chain Y's off-chain balance. The Chain Y operator mints the funds after receiving this Deposit attestation.

6. After the `GatewayMint` event, Circle's backend service burns the corresponding funds held for the Chain X depositor address in `GatewayWallet`.

Whenever a token is added to the list of supported tokens, the `xReserve` contract gives infinite allowance to `GatewayWallet` and `TokenMessenger`. Moreover, anyone can call the external method `setUnlimitedAllowances()` on xReserve to give infinite allowance to aforementioned contracts for user specified tokens. This reverts if one of the user specified tokens is not previously supported.

2.2.6 Details About BurnIntent and Attester Rotation

As previously described, a chain operator sends a signed `BurnIntent` to Circle's backend. This `BurnIntent` is submitted together with a set of signatures. Each signature in the set is signed by an attester, where attesters are enabled or disabled by the domain manager for the remote chain.

Either Circle's backend directly interacts with the `GatewayWallet` or the chain operator calls `xReserve.withdraw()`. In both cases, `GatewayWallet.gatewayBurn()` is invoked with the intent and the signatures, which calls `RemoteDomainDepositor.isValidSignature()` to validate the signatures. This function returns true if the set contains a *required* number of signatures and each signature is from a distinct enabled attester.

The *required* number of signatures is called the threshold, which can be changed by the domain manager. A delay is in place before a threshold change is fully activated. During this delay period, both the old and the new threshold values are accepted. Similarly, when the domain manager disables an attester, the attester is fully disabled only after a delay. During the delay period, signatures from that attester are still accepted.

2.2.7 Deposit Hook System for Chain Partners

Chain partners can deploy custom `IRemoteDomainHookExecutor` contracts and register them on the `xReserve` contract with Circle's cooperation to enforce additional validation checks or custom execution logic. When `depositToRemote()` is called, if a `hookExecutor` exists for the remote domain, `executeHook(hookData)` is invoked. Hook executors must be carefully validated before being registered. The security of these executors is out of scope for this report.

2.2.8 Version 4

In Version 4, deposits using permits (`depositToRemoteWithPermit`) and authorizations (`depositToRemoteWithAuthorization`) have been removed from the codebase.

2.3 Trust Model

2.3.1 Roles on Smart Contract Level

The roles for `xReserve` are as follows:

- **owner**: Trusted; they can add/remove any token to the list of supported tokens. They can upgrade the implementation for `xReserve` to any address, and set the regisrationManager to any address. Furthermore, for any registered domain they have the following capabilities:
    - deregistering the domain
    - setting a remote domain hook executor for the domain
    - deregistering any remote token on the domain
    - setting the domain manager to any address
    - setting the persistent signature buffer delay to any non-zero value

- **pauser**: Semi-trusted; they are the global pauser and can pause/unpause the contract to allow/disallow deposits and withdrawals.
- **blocklister**: Semi-trusted; they can blocklist an address on a remote domain, preventing it from participating in cross-domain operations.
- **registrationManager**: Semi-trusted; they can register any domain and any remote token.

The roles for `RemoteDomainDepositor` are as follows:

- **domainManager**: Semi-trusted; they can set the domainPauser to any address, enable/disable attesters and set the signature threshold for their respective domain.
- **domainPauser**: Semi-trusted; they can pause/unpause withdrawals as well as deposits for their respective remote domain. Deposits to the remote domain can be paused without pausing withdrawals, and vice versa.

2.3.2 Chain Partners

Circle must either trust their chain partners or keep track of mints/burns on remote chains. There are two potential scenarios where a chain partner can affect the total supply of USDC:

- In case a chain operator sends a signed BurnIntent to Circle without a corresponding burn of USDC-backed stablecoin on that chain, the USDC-backed stablecoin won't be fully backed by USDC in xReserve contract.
- In case a chain operator does not mint USDC-backed stablecoin upon receiving a Deposit Attestation, USDC locked in GatewayWallet for the remote blockchain will be higher than the USDC-backed stablecoin on the remote blockchain.

Any external token used by the system is considered fully trusted and should be carefully assessed before being added to the allowlist. We assume only ERC20-compliant tokens without special behavior such as inflationary/deflationary tokens, delayed finality, tokens with transfer hooks, fees on transfer, tokens that transfer less than the specified amount, etc., are used with this system.

`xReserve` should not have any elevated privileges for CCTP or Gateway, as it invokes CCTP or Gateway methods via hooks.

2.3.3 Off-Chain Verification

Off-chain components of the system must be fully trusted to enforce all the required checks, see the details provided in On-chain and off-chain checks.

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:

- 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	1

- Insufficient Guarantee on Absent Address Risk Accepted

5.1 Insufficient Guarantee on Absent Address

Correctness Low Version 1 Risk Accepted

CS-CRCLxRes-001

The function `_bytes32ToAddressSafe` in `AddressLib` reverts if `uint256(buf) >> 160 != 0`. However, this does not guarantee that there is no incorrectly formatted address hidden in the `bytes32`. Consider the 20-byte address `0x00...00AB00..00` with 12 leading `00` bytes and 7 trailing `00` bytes. If the backend right-pads this address, `_bytes32ToAddressSafe` will not fail and instead return `0xAB000..00`.

Risk accepted:

Circle has decided to take no further action. Stating it is acceptable, as there is no reliable way to distinguish between an incorrectly formatted address and a valid address containing many zeros, and preventing such cases is not within the function's intended scope.

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	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Redundant Check Code Corrected

Informational Findings	4

- Duplicated Code Code Corrected
- Event Indexing Code Corrected
- Missing Zero Address Check Code Corrected
- Unused Function Code Corrected

6.1 Redundant Check

Design Low Version 1 Code Corrected

CS-CRCLxRes-002

In `Attestable` the function `disableAttester` checks if the number of persistent enabled attesters is less than the minimum signature threshold and then if it is less than the current signature threshold. However, `MIN_SIGNATURE_THRESHOLD` must always be smaller than the current signature threshold, which makes the first check redundant.

The threshold must be larger because when setting the threshold, `_validateSignatureThreshold` is called, which will revert if the threshold is less than the minimum signature threshold.

Code corrected:

The first check has been removed.

6.2 Duplicated Code

Informational Version 1 Code Corrected

CS-CRCLxRes-004

1. In `NoValidationAttestationLib._asAttestationOrSetView` the function must handle either a single attestation or an attestation set. The function `_isSet` does check this. However, the function `_asAttestationOrSetView` does not use `_isSet` but reconstructs the same check the other way around by checking if the magic value indicates a single attestation to branch off.

2. In `Blocklistable.updateBlocklister` and in `__Blocklistable_init` the event `BlocklisterUpdated` is emitted. It could be emitted within the function `_setBlocklister` to avoid code duplication.

Code corrected:

`_asAttestationOrSetView` uses `_isSet` now and the event is emitted within `_setBlocklister`.

6.3 Event Indexing

Informational **Version 1** **Code Corrected**

CS-CRCLxRes-005

1. The event `TokenSupported` in `TokenSupport` emits the `token` parameter. It could be indexed for better searchability.
2. Considering that deposits and withdrawals are complementary actions, and that the event `Withdrawn` indexes the parameter `localRecipient`, the event `DepositedToRemote` could index the `remoteRecipient` parameter.

Code corrected:

Both parameters are indexed in the updated code.

6.4 Missing Zero Address Check

Informational **Version 1** **Code Corrected**

CS-CRCLxRes-011

In `RemoteDomainRegistration.__RemoteDomainRegistration_init()` the `registrationManager_` parameter is assigned to state via `_setRegistrationManager()` without being checked for `address(0)`. While `updateRegistrationManager` performs this check, the initializer does not.

Code corrected:

The `_setRegistrationManager` function now checks for `address(0)`.

6.5 Unused Function

Informational | Version 1 | Code Corrected

CS-CRCLxRes-017

The `AddressLib` library contains a function `_addressToBytes32` that is only used in tests and might unnecessarily bloat the bytecode in deployments.

Code corrected:

The function `_addressToBytes32` has been removed.

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 Constant Missing Explicit Visibility

Informational | Version 1 | Acknowledged

CS-CRCLxRes-003

The following contracts have constants without an explicit visibility declaration:

1. In `Constants`
2. In `WithdrawHookData`
3. In `DepositIntent`

Acknowledged:

Circle states that the variables are not intended to be exposed at the contract level, but are instead provided to enhance code readability.

7.2 Gas Optimization

Informational | Version 1 | Code Partially Corrected

CS-CRCLxRes-006

1. In `RemoteDomainRegistration.setDomainPauseState` the condition `_domainPauser == address(0) || msg.sender != _domainPauser` is used. However, `msg.sender` cannot be the zero address, which makes the first check redundant.
2. `isAttesterEnabled` is only used once in `Attestable._isValidSignatureHelper`. `_isValidSignatureHelper` reads `$.persistentEnabledAttesters.contains(_recoveredAttester)` before calling `isAttesterEnabled`. In `isAttesterEnabled` the same read is performed again.
3. `Blocklistable.__Blocklistable_init` reads `oldBlocklister` from storage. As this is during initialization, the value must be zero, making the `SLOAD` unnecessary.
4. In `Attestable` the function `disableAttester` checks if the number of persistent enabled attesters is less than the minimum signature threshold. However, `MIN_SIGNATURE_THRESHOLD` must always be larger than the current signature threshold, which makes the first check redundant.
5. In `Attestable.disableAttester` the storage variable `data.persistentEnabledAttesters` is accessed three times, resulting in redundant `SLOAD` operations.

Code partially corrected:

1. The address zero check has been removed.
2. Kept as is.

3. Still reads zero for `oldBlocklister` but the function was refactored to emit the event in the internal function.
4. The redundant check was removed.
5. Rearrangement was performed.

7.3 Inconsistent State Transition

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

CS-CRCLxRes-008

If `RemoteDomainRegistration.deregisterRemoteDomain` is called, the `localTokenToRemoteTokenMapping` may still be set if the deregistration is not done in the correct order. This would lead to an inconsistent state, and if the same domain is re-registered, the old mapping would still be present.

Acknowledged:

Circle is aware and states they will ensure the correct steps will be taken to prevent this scenario from happening.

7.4 Incorrect or Missing Comments and Natspec

Informational **Version 1** **Code Partially Corrected**

CS-CRCLxRes-009

1. In `NoValidationAttestationLib._asAttestationOrSetView` the first dev comment `Reverts with 'InvalidTransferPayloadMagic' if neither known magic number is present` is incorrect.
2. In `Attestable.attesterValidUntilBlock` the notice is misleading, as an address that was never an attester also has value 0.
3. In `Attestable._setSignatureThreshold` there are two examples of how to treat signature threshold updates. However, the examples are incorrect as they state: signatures with `4 or 5` attesters will be rejected after the delay, and signatures with `3 or 4` attesters will be rejected after the delay, respectively. This is not correct, as it is not a range. In both cases, `4` signatures are always rejected due to the equality check.
4. In `RemoteDomainRegistration` the internal function `__RemoteDomainRegistration_init` is defined in the section for `External Functions`.
5. In `Attestable` the Natspec for the error `SignatureThresholdUpdateInProgress` states: `validUntilBlock The block number until which the pending update remains valid`. It would be the block number until which the old signature threshold remains valid.

Code corrected:

1. The dev comment has been corrected.
2. The notice has been corrected.
3. The comment was kept as is.
4. The internal function has been moved accordingly.

5. The Natspec for the error `SignatureThresholdUpdateInProgress` has been corrected.

7.5 Missing Event

Informational | Version 1 | Acknowledged

CS-CRCLxRes-010

The function `_setUnlimitedAllowances` could emit an event when it sets the unlimited allowances.

Acknowledged:

Circle acknowledges that the event is not necessary and has decided to take no further action.

7.6 No Option to Revoke Supported Tokens

Informational | Version 1 | Acknowledged

CS-CRCLxRes-012

The `TokenSupport` contract does not provide a function to revoke supported tokens. Without clear specification, we cannot verify if this is intended or has been forgotten.

Acknowledged:

Circle acknowledges that the function to revoke supported tokens is not necessary. It is by design and consistent with the gateway contracts. Hence, Circle has decided to take no further action.

7.7 No Upper Bound on Signers

Informational | Version 1 | Acknowledged

CS-CRCLxRes-013

The contract does have a minimum of two signers, but does not have an upper bound on the number of signers. It might get costly and in a very theoretical situation could cause out of gas issues, ultimately leading to a revert in `_isValidSignatureHelper`.

Circle might evaluate if an upper bound on the number of signers makes sense to be implemented.

Acknowledged:

Circle acknowledges that and opted to enforce an upper bound on signers off-chain to allow for more flexibility, since the reasonable maximum number of signers may differ for different EVM chains.

7.8 Non-recoverable Tokens or Native Tokens

Informational | Version 1 | Acknowledged

CS-CRCLxRes-014

In case a user unintentionally sends ETH or other tokens to the bridge contract, the tokens will be stuck in the contract as there is no sweep or recover function. However, the contract could be updated to include one. Circle might evaluate if a recover function should be implemented.

Acknowledged:

Circle is aware and states that this is by design.

7.9 Outdated Dependencies

Informational | Version 1 | Acknowledged

CS-CRCLxRes-015

The latest version of `openzeppelin-contracts` and `openzeppelin-contracts-upgradeable` is 5.4.0. The codebase currently uses version 5.0.1 of `openzeppelin-contracts` and version 5.3.0 of `openzeppelin-contracts-upgradeable`.

Acknowledged:

Circle acknowledges the issue and keeps the code as it is.

7.10 Spurious Events

Informational | Version 1 | Acknowledged

CS-CRCLxRes-016

1. In `Blocklistable` the functions `blocklist`, `unblocklist` and `updateBlocklister` will emit an event even in case of re-setting the same value.
2. In `Pausing` the function `_setDomainPauseState` and `_setPauser` will emit a spurious event even in case of re-setting the same value.
3. In `RemoteDomainRegistration` the function `setRemoteDomainHookExecutor` and `updateRegistrationManager` will emit a spurious event even in case of re-setting the same value.

Acknowledged:

Circle acknowledges the behavior and keeps the code as it is.

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 Bypassing xReserve Pause

Note Version 1

Even when `xReserve` is paused, a chain operator's valid burn intent can still be used directly with `GatewayWallet`.

8.2 Hooks Can Disable All Deposit Flows

Note Version 1

Circle must ensure that a registered `remoteDomainHookExecutor` is not malicious. For example, the `executeHook()` method could be implemented to always revert. This would cause every deposit flow to fail.

8.3 On-chain and Off-Chain Checks

Note Version 1

Bridges are delicate systems that heavily rely on the off-chain system. The scope of this audit is the smart contracts. However, it is impossible to assess whether the contracts provide the security guarantees without knowing which checks are performed off-chain or knowing when all relevant calls are made and in what order.

To provide the opportunity to self-check the off-chain system, the following non-exhaustive list suggests what checks should be performed off-chain in addition to the checks required to be performed in the Gateway system is provided:

1. The backend must pick up the `DepositedToRemote` event regularly and process it considering block finality thresholds.
2. The backend must issue a valid deposit attestation for each event and should ensure it is picked up and executed on the remote chain by the operator.
3. Objects must be well-formatted including the correct MAGIC value.
4. All fields must be validated to match the on-chain event.
5. Nonce management must be done correctly and validated if progressed correctly.
6. The signature must be checked, and all relevant domain information must be included.
7. The hookdata length should have boundary checks (hookdata should also be valid and non-reverting).
8. The backend must be able to pick up burn intents from the operators.
9. The burn intent must be validated with respect to Gateway's protocol rules. A pure signature check is not sufficient. All fields must be checked, including the salt and the signature's validity according to the Gateway protocol.

10. The signature must be checked against the on-chain attestation set correctly via `isValidPersistentSignature`.
11. The backend must execute the burn before the attester set or threshold update delay `persistentSignatureBufferDelayBlocks` has passed.
12. A valid mint attestation must be made available to the operator.
13. Off-chain balances must be accounted for correctly.

8.4 Padding in Bytes32 to Address Conversion

Note Version 1

The function `AddressLib._bytes32ToAddressSafe` is used to convert a `bytes32` to an address. As it cannot be reviewed or guaranteed that the value is constructed correctly, we would like to point out that it has to be left-padded with zeros. This is not the common way to construct a `bytes32`.

8.5 Using Draft or Stagnant Standards

Note Version 1

The code references or uses certain standards that must be handled carefully, especially when they are marked as draft, such as ERC 7597 and 7598. ERC 3009 is even marked as stagnant.