PUBLIC

Code Assessment

of the Circle Gateway

Smart Contracts

July 08, 2025

Produced for

CIRCLE

by

CHAINSECURITY

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 Gateway according to Scope to support you in forming an opinion on their security risks.

Circle implements a set of smart contracts for Circle Gateway, a crosschain primitive that enables the bridging and aggregation of user's liquidity on any target chain with low latency.

The most critical subjects covered in our audit are functional correctness, signature handling, and access control. Security regarding all the aforementioned subjects is high. All issues identified in the intermediate reports have been resolved.

The general subjects covered are front-running, and gas efficiency. Some user operations may revert due to front-running, see Deposit with Permit / Authorization Is Susceptible to Griefing Attacks.

In summary, we find that the codebase provides a good 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	2
• Code Corrected	1
• Specification Changed	1
Medium-Severity Findings	0
Low-Severity Findings	7
• Code Corrected	4
• Specification Changed	2
• 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 Gateway repository based on the documentation files. Additionally, the relevant functions used from the `TypedMemView` library contract were assessed.

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	29 Apr 2025	f24a7d19bf6f9458443991961a174de87edeee91	Initial Version
2	05 Jun 2025	ca774f13f51d924c58d48d40ba6a5735370af26b	Version 2
3	24 Jun 2025	5b5446f5c622901acaea6a875b022425eecb0c13	Version 3

For the solidity smart contracts, the compiler version `0.8.29` was chosen and `evm_version` is set to `cancun`. The following files were in the scope of this review:

```
lib/
    AddressLib.sol
    AttestationLib.sol (named MintAuthorizationLib.sol in Version 1)
    Attestations.sol (named MintAuthorizations.sol in Version 1)
    BurnIntentLib.sol (named BurnAuthorizationLib.sol in Version 1)
    BurnIntents.sol (named BurnAuthorizations.sol in Version 1)
    Cursor.sol (named AuthorizationCursor.sol in Version 1)
    EIP712Domain.sol (introduced in Version 2)
    TransferSpec.sol
    TransferSpecLib.sol
modules/
    common/
        Denylist.sol
        Domain.sol
        Pausing.sol
        TokenSupport.sol
        TransferSpecHashes.sol
    minter/
        Mints.sol
    wallet/
        Balances.sol
        Burns.sol
        Delegation.sol
        Deposits.sol
        WithdrawalDelay.sol
        Withdrawals.sol
GatewayCommon.sol
GatewayMinter.sol
GatewayWallet.sol
UpgradeablePlaceholder.sol
```

2.1.1 Excluded from scope

All third-party code like libraries are excluded from the scope, except for the relevant functionalities from the `TypedMemView` library that are used in the codebase. The system relies heavily on off-chain operators which are not in scope of this review.

As a crosschain 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. The same applies for supported tokens to be instant-trasferred as some might be incompatible with the current setup (e.g., re-basing tokens, tokens with fees on transfers, etc.).

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 a set of smart contracts for the Gateway system, a crosschain primitive that enables the transfer and aggregation of user's liquidity on any target chain with low latency. This system is designed to be non-custodial: Users can withdraw their funds without the Gateway system's cooperation, although this is subject to a time delay.

In general the system works as follows:

1. User deposit funds to a GatewayWallet contract on any supported chain and wait for block finality.
2. Gateway system issues an Attestation (named Mint Authorization in **Version 1**) to the user after the user provides a signed BurnIntent (named Burn Authorization in **Version 1**).
3. User consumes the signed Attestation on the destination chain's GatewayMinter to mint tokens.
4. Gateway system consumes the signed BurnIntent on the source chain to burn tokens from the GatewayWallet. A fee might be charged for the service.

2.2.1 Deposits

GatewayWallet works as an escrow that accepts user deposits with following interfaces:

- `deposit()`: A simple deposit that requires prior allowance granted.
- `depositWithPermit()`: The token is assumed to be ERC-7597 compliant, which accepts a permit signature to increase the allowance. Note, custom signature verification is supported if the signer is a contract.
- `depositWithAuthorization()`: The token is assumed to be ERC-7598 compliant, which supports transfer with signed authorizations. Note, custom signature verification is supported if the signer is a contract.

Only whitelisted tokens can be deposited and all the parties (`msg.sender` and depositor) involved are required to be not denylisted. Deposits are blocked if the contract is paused.

2.2.2 Minting And Burning

After the transaction of the token deposits has been finalized on the source domains, users can instant-transfer the tokens to another supported domain with the assistance of the Gateway system:

- User must sign a BurnIntent that allows the Gateway system to burn the deposited funds on the source domain, a fee might be charged for the service.
- Upon receiving the signed BurnIntent, the Gateway system will issue an Attestation with its `attestationSigners`, which allows the user to mint tokens on the destination domain's GatewayMinter contract.
- If a Attestation is consumed by the user on the destination domain, the Gateway system's `burnSigners` will sign the respective BurnIntent. By submitting both the user's and the `burnSigner`'s signatures to the GatewayWallet contract, the Gateway system can burn user's funds on the source domain.
- Minting on the same domain is supported. Since Version 2, same-chain transfers are implemented similar to cross-chain transfers: tokens are first minted and burned at a later point in time. Therefore, the total supply of the token is inflated after the minting happens. In Version 1, same-chain transfer would move tokens from the GatewayWallet to the recipient instead.

Both the BurnIntent and the Attestation contain the same *TransferSpec* that embeds the detailed transfer information. The BurnIntent must have a significantly longer expiration time (e.g., 3 days) than the Attestation (e.g., few minutes).

In more details:

The function `GatewayMinter.gatewayMint()`:

1. Accepts one or a set of Attestation with a valid signature from the Gateway system's `attestationSigners`.
2. It performs sanity checks of each Attestation, most importantly, checks expiration; the Attestation is targeted for the GatewayMinter contract on the local domain; and the `msg.sender` is not denylisted.
3. An extra check on the `sourceToken` and `destinationToken` is performed if the transfer is in the same domain.
4. Eventually it will mint tokens to the recipient.

The function `GatewayWallet.gatewayBurn()`:

1. Accepts one or a set of BurnIntent from different users with the respective user signatures and actual fees charged.
2. The submitted data `(intents, signatures, fees)` must be validated and signed by the Gateway system's `burnSigner`.
3. The (set of) BurnIntent(s) of each user will be checked and processed:
    1. If the BurnIntent is not targeted at this domain (or a same domain transfer in Version 1), it will be skipped.
    2. The BurnIntent is not expired; the source domain and contract match; the token is supported; and the actual fee is below `maxFee` provided by the user.
    3. The signature is valid and the recovered signer matches with the `sourceSigner`.
    4. The signer was ever authorized for the source depositor.
4. It is required that all the relevant BurnIntents in a set use the same burn token address.
5. Eventually the token is burnt by reducing the available (and withdrawing) balance of the depositor. The fee is minted to the `feeRecipient`.

Each Attestation and BurnIntent can only be processed once since the `TransferSpecHash` prevents replaying the same Attestation or BurnIntent. The view function `isTransferSpecHashUsed()` returns the status of a `TransferSpec`. GatewayMinter should have the required role to call `mint()` on all the supported token (or MintAuthority), while GatewayWallet should have the required role on tokens to call `burn()`.

2.2.3 Withdrawals

A two-step withdrawal approach is implemented to withdraw funds from the Gateway Wallet. This is mostly intended as fallback for users to retrieve their funds if the Gateway system's services issuing Attestations are not usable, to ensure the non-custodial property of the system.

1. `initiateWithdrawal()`: User has to initiate a withdrawal first, which will update the internal accounting of the available balance and the withdrawing balance. The `withdrawalBlock` will be set to `withdrawalDelay` from current block number.
2. `withdraw()`: Once `withdrawalDelay` has elapsed, user can finalize the withdrawal and all the withdrawing balance will be transferred to the recipient.

Note, initiating a new withdrawal will always extend the `withdrawalDelay` of the existing pending withdrawals.

2.2.4 Delegates

Delegates are roles who can operate on behalf of the users (depositors). Users can configure delegates with:

- `addDelegate()`: Add a new delegate to operate on behalf of the user for a specific token.
- `removeDelegate()`: Revoke an existing delegate.

The main responsibility of delegates are:

- Initiate or finalize withdrawals on behalf of users (this functionality was removed in **Version 3**).
- Sign BurnIntent on behalf of users. **Note**, even if a delegate is revoked, its future signatures for a BurnIntent are still considered valid by the smart contracts.

The Gateway Wallet and Minter are expected to be deployed behind a proxy and the implementation uses UUPSUpgradeable pattern where the upgrade is restricted to the owner. They further inherit Pausing and Denylist:

- Deposit, withdraw, mint and burn can all be paused.
- A denylisted address can withdraw, however, cannot further deposit, burn or mint.

2.3 Changes in Version 2

The codebase was refactored in **Version 2**, and the following changes were implemented:

- A new function `depositFor` was added in contract Deposits. The caller provides the tokens required for the deposit, but they are accounted on the balance of `depositor` specified by the caller.
- The hashing and signing of burn intents is made compliant with EIP-712.
- The fee behavior is now consistent for both same-chain and cross-chain transfers.
- The supported tokens added in the GatewayWallet should implement the interface `IBlacklistableToken`.
- A mapping is now used to whitelist accounts with roles `attestationSigners` and `burnSigners`. This approach facilitates the rotation of accounts that have any of the signing roles.

2.4 Changes in Version 3

- Only depositors can initiate and withdraw funds from the GatewayWallet.

2.5 Trust Model

Owner: owners of both GatewayMinter and GatewayWallet are fully trusted; they have privileges to:

1. Upgrade the implementation and add the supported tokens. These changes are not subject to delays, hence `owner` can break the non-custodian property of the system by upgrading to a different implementation contract.
2. Setting denylister, pauser, mint authority.
3. Gateway Minter's owner can update `attestationSigners`, and Gateway Wallet's owner can update the `burnSigners`, `feeRecipient`, and `withdrawalDelay`.

If `owner` is compromised, they can upgrade the contract to a malicious implementation to drain deposited funds or mint new tokens.

Pauser: trusted; it can pause the main functionalities of the contract. If this account is compromised, it can cause denial-of-service (including withdrawals), but it cannot access user's funds.

Denylister: trusted; it can block users from using the crosschain primitive. If this account is compromised, it can cause denial-of-service by denylisting core contracts, but it cannot access user's funds.

Attestation signer: fully trusted; if any account with this role is compromised, the attacker can exploit the minting role of this contract to mint supported tokens.

Burn signer: trusted; they should validate that the encoding of the calldata they sign is correct. Accounts with this role cannot directly access user's funds but they can enable double spending by not signing burn intents. Furthermore, Burn signers and Attestation signers should always be distinct accounts.

Delegatees: fully trusted by the delegator (user). Delegatees can reset the timer for withdrawing delay, instant-transfer delegator's funds to arbitrary addresses or withdraw them from the wallet.

Users: not trusted.

Furthermore, any external token used by the system is considered fully trusted and should be carefully assessed before being whitelisted as a supported token. We assume only ERC20-compliant tokens without special behavior (e.g., inflationary/deflationary tokens, delayed finality, transfer hooks, fees on transfer, etc.) and implementing `IMintBurnToken`, `IBlacklistableToken`, `IERC7597` and `IERC7598` interfaces are supported by the system. Finally, supported tokens should also implement the function `burn` which reverts on failure.

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	1

- Deposit With Permit / Authorization Is Susceptible to Griefing Attacks Risk Accepted

5.1 Deposit With Permit / Authorization Is Susceptible to Griefing Attacks

Design Low Version 1 Risk Accepted

CS-CSpend-003

The external functions `depositWithPermit()` and `depositWithAuthorization()` in the contract Deposits are susceptible to griefing attacks. Attacker front-runs a legit transaction and calls `permit()` / `receiveWithAuthorization()` function of the target token to consume the nonce. Therefore, the respective call to Deposits contract may revert due a used nonce.

Risk accepted:

Circle is aware and accepts the risk. Since the contract cannot distinguish between an already used permit and an invalid one, it requires the permit to be valid. Allowing the deposit to proceed using existing allowance would expose users to worse risks such as unauthorized fund movement (i.e. any user who has approved *GatewayWallet* for their USDC).

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

- Possible to Inject Data on Burn Authorizations Code Corrected
- Violation of Non-Custodial Property Specification Changed

Medium-Severity Findings	0

Low-Severity Findings	6

- Same Chain Transfers May Be Subject to a Fee Specification Changed
- Unchecked Return Value of Identity Precompile Code Corrected
- Function Implementation Does Not Match Specifications Specification Changed
- Inconsistent Behavior of View Functions in Balances Code Corrected
- Indexed Addresses in DenylisterChanged Code Corrected
- Non-standard EIP-7201 Formula Identifier Code Corrected

Informational Findings	7

- Missing Blacklist Check Specification Changed
- Unused Import in Balances Code Corrected
- EIP-1155 Compliant View Functions in Balances Might Revert Code Corrected
- Misleading Field Name Nonce Code Corrected
- Missing Sanity Checks Code Corrected
- Unused Functions Code Corrected
- Validation of TypedMemView References Specification Changed

6.1 Possible to Inject Data on Burn Authorizations

Security High Version 1 Code Corrected

CS-CSpend-001

The function `gatewayBurn()` is called to burn deposited funds from the GatewayWallet contract by providing a signature from `burnSigner` over the calldata body that embed the `(BurnAuth, signatures, fees)`.

```
function _verifyBurnerSignature(bytes memory burnerSignature)
    internal view {

    ...
    bytes memory calldataBytes = msg.data[0x84:msg.data.length - 0x80];

    // Verify the signature and revert if it's invalid
    address recoveredSigner = ECDSA.recover(
        keccak256(calldataBytes).toEthSignedMessageHash(), burnerSignature);
    if (recoveredSigner != BurnsStorage.get().burnSigner) {
        revert InvalidBurnSigner();
    }
}
```

The signature from `burnSigner` is verified against the `msg.data` body. However, the encoding of calldata could be variable; and a customized calldata encoding can be exploited to inject malicious payload and bypass the burner signature check. Consider the following attack scenario (also illustrated in the plot):

1. User signs a BurnAuth (with expiry in 3 days) and receives a MintAuth (with expiry in 2 minutes) to mint 10k USDC on Base.
2. User eventually decides to not transfer the funds, hence the MintAuth expires after 2 minutes.
3. Attacker learns user's (BurnAuth, BurnSig), and then attaches them into the metadata of his own BurnAuth.
4. Now, the attacker follows the typical workflow for his own Gateway transfer (also shown in the plot):

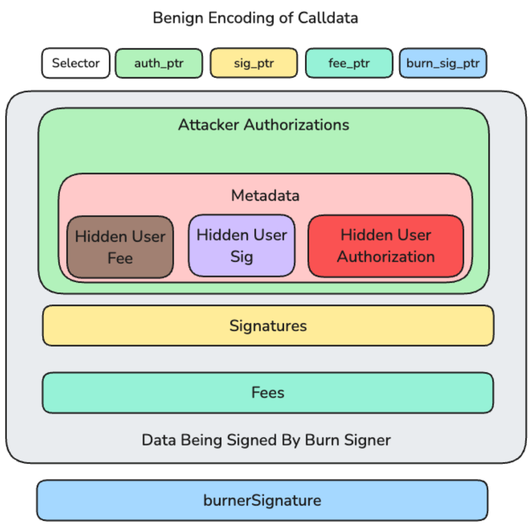
    4.1. He signs the BurnAuth to get a MintAuth to mint 10 USDC on Base.

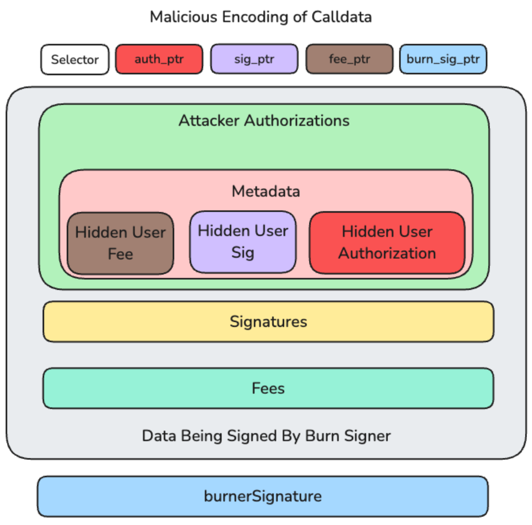
    4.2. He consumes the MintAuth on Base.

    4.3. Burn signer signs the calldata to burn attacker's 10 USDC onchain.

    4.4. Attacker calls `gatewayBurn()` with the calldata signed by `burnSigner` but only changes the first 3 offsets, which now points to the attackers's (BurnAuth, BurnSig, Fee) in the metadata of the attacker's `TransferSpec`. As a result, the validation of the signature from `burnSigner` passes, and the user's 10k USDC will be burned on the source chain, even though the respective MintAuth is not used.

Note in this attack scenario, the attacker can also freely choose any fees when burning user's funds. This also enables attackers to consume their burn authorizations with zero fees.





In summary, the attacker can:

- Execute other user's BurnAuth that should not be executed, leading to loss of user funds.
- Execute other user's BurnAuth with maximal fees.
- Execute their own BurnAuth with zero fees.

Code corrected:

The external function `gatewayBurn()` has been revised to remove the usage of `msg.data` that enabled the attack presented above. The function now takes two inputs, a `bytes` array which represents an ABI-encoded calldata, and the burner signature:

```
function gatewayBurn(bytes calldata calldataBytes, bytes calldata signature) external whenNotPaused {
    // Verify that the calldata was signed by a valid burn signer
    _verifyBurnSignerSignature(calldataBytes, signature);

    // Decode the calldata into the intents, signatures, and fees arrays
    (bytes[] memory intents, bytes[] memory signatures, uint256[][] memory fees) =
        abi.decode(calldataBytes, (bytes[], bytes[], uint256[][]));

    ...
}
```

The code now decodes `intents`, user `signatures`, and `fees` from the the same `calldataBytes` that are signed by the trusted account.

6.2 Violation of Non-Custodial Property

Correctness High Version 1 Specification Changed

CS-CSpend-002

The contract GatewayWallet is designed to be non-custodial, meaning that privileged roles such as `burnSigner` or `mintAuthorizationSigner` cannot access users' deposited funds. GatewayWallet requires a BurnAuth signature from the `depositor` or their `delegatee` in order to reduce the user's balance.

However, this property does not hold for same-chain transfers. In this case, the call path `gatewayMint()` reduces the user's balance but does not require any signature from the user. As a result, it allows the privileged role `mintAuthorizationSigner` to freely transfer the user's funds to arbitrary recipients.

Specification changed:

The same-chain transfers have been refactored in Version 2 to follow the instant-transfer approach instead of simple transfers. This means same-chain transfers are made in line with cross-chain transfers. Therefore, the functions `Mint._mintOrTransfer()` and `Burns.gatewayTransfer()` have been removed.

6.3 Same Chain Transfers May Be Subject to a Fee

Correctness Low Version 2 Specification Changed

CS-CSpend-013

The code did not allow to take a fee from same chain transfers in version 1 of the code in line with the documented product requirements:

```
Same-chain withdrawals are the same as cross-chain instant transfers, except the funds are
transferred out of the wallet contract instead of being minted and then burned.
No fee is charged for this, since the Gateway system does not have to cover the gas fee for
the burn.
```

This part of the code was refactored as part of the fix of issue Violation of non-custodial property. Since same chain transfers are now using the same functionality as cross chain transfers, the Gateway system has to burn these tokens. The statement in the text is no longer correct and the code now allows to take fees.

Specifications changed:

The inline NatSpec comments of the contract GatewayWallet have been revised to clarify that a fee is charged also for same-chain transfers:

```
/// @notice For same-chain withdrawals, ...
/// A fee is deducted from the user's balance within the `GatewayWallet` contract
/// in addition to the requested amount, since Circle incurs gas costs
/// for the burn operation.
```

6.4 Unchecked Return Value of Identity Precompile

Design | Low | Version 2 | Code Corrected

CS-CSpend-014

The function `TransferSpecLib.getTypedDataHash()` performs an external call to the identity precompile to copy data into memory:

```
// The pop() removes the success boolean returned by staticcall since we don't need it.
pop(staticcall(gas(), 4, footerStart, footerLen, add(ptr, 128), footerLen))
```

The code, as suggested by the inline comment, expects that the call to the identity precompile always succeeds, however the call can fail due to insufficient gas.

Code corrected:

The success flag returned by the identity precompile is now checked:

```
let success := staticcall(gas(), 4, footerStart, footerLen, add(ptr, 128), footerLen)
if iszero(success) {
    // Revert with custom error
    ...
}
```

if the success flag is false, the function `getTypedDataHash()` reverts with a custom error.

6.5 Function Implementation Does Not Match Specifications

Correctness Low Version 1 Specification Changed

CS-CSpend-004

The NatSpec description for function `validateBurnAuthorizations` states:

```
Allows anyone to validate whether a set of burn authorizations would be valid if it
were signed by the specified signer (which must match `sourceSigner` in the
TransferSpec`).
```

However, the implementation of the function lacks important checks that are performed when a BurnAuth is executed. For instance, `validateBurnAuthorizations()` uses a fee of zero but a non-zero fee might be charged when BurnAuth is consumed. The function does not check if the TransferSpec has already been consumed. Moreover, it does not check if the depositor has enough balance and if there is an actual balance change. Therefore, even if `validateBurnAuthorizations()` returns `true` and a valid signature from `signer` is provided, it does not guarantee that the BurnAuth will execute successfully on-chain.

Specification changed:

The view function has been removed in Version 2 of the codebase.

6.6 Inconsistent Behavior of View Functions in Balances

Design Low Version 1 Code Corrected

CS-CSpend-005

The majority of view functions in the Balances contract use the modifier `tokenSupported` which reverts if the input `token` is not supported. However, functions `totalBalance()` and `availableBalance()` do not use this modifier.

Code corrected:

The check if the input token is supported has been removed from view functions `withdrawingBalance()`, `withdrawableBalance()`, `balanceOf()` and `balanceOfBatch()`, which now return `0` instead of reverting.

6.7 Indexed Addresses in DenylisterChanged

Design Low Version 1 Code Corrected

CS-CSpend-016

The `DenylisterChanged` event indexes its arguments, whereas other similar events emitted during role changes do not.

Code corrected:

The events emitted when roles changes have been updated to be consistent.

6.8 Non-standard EIP-7201 Formula Identifier

Design Low Version 1 Code Corrected

CS-CSpend-006

The EIP-7201 specifies the formula identifier `erc7201` to be used in the NatSpec tag of the struct type, however the codebase uses the identifier `7201`:

```
/// @custom:storage-location 7201:...
```

Code corrected:

The formula identifier has been fixed in the NatSpec tag of the struct types.

6.9 Missing Blacklist Check

Informational Version 2 Specification Changed

CS-CSpend-018

The supported tokens, e.g. USDC, feature an address blacklist. If one deposits into a gateway wallet and initiates a withdrawal, anytime after the withdrawal delay has passed one can withdraw and specify any fresh address as the recipient of the tokens.

This lets one park funds without risk of being blacklisted. Once the holder decides to access them, he picks a clean address and withdraws, then quickly uses or converts the funds before the Gateway system can freeze them. There’s still some risk for the malicious token holder if the Gateway system upgrades the contract, but it's a simple way to park such funds.

This is not limited to this system, but a general pattern possible in systems where one contract holds all funds centrally and users are able to withdraw later.

However since this system is operated by Circle, if this happens it may be interpreted negatively.

Specification changed:

In Version 3, delegatees cannot initiate or complete a withdrawal anymore. Furthermore, the function `withdraw()` always transfers funds to the `msg.sender` which should be the `depositor` address. Therefore, if the `depositor` is blacklisted on USDC, the transfer will revert. This implies that any integrating 3rd party smart contract must be able to call `initiateWithdrawal()/withdraw()` itself.

6.10 Unused Import in Balances

Informational Version 2 Code Corrected

CS-CSpend-015

The contract `Balances` does not use the modifier `tokenSupported` in Version 2, hence the contract `TokenSupport` is imported but not used.

Code corrected:

The unused import has been removed.

6.11 EIP-1155 Compliant View Functions in Balances Might Revert

Informational | Version 1 | Code Corrected

CS-CSpend-017

The view functions `balanceOf()` and `balanceOfBatch()` in the contract `Balances` revert if the data encoded in the input `id` is invalid, i.e., `token` is not supported, or the `balanceType` is invalid.

Code corrected:

The view functions `balanceOf()` and `balanceOfBatch()` do not revert anymore if the token is not supported or the `balanceType` is invalid, but return `0` instead.

6.12 Misleading Field Name Nonce

Informational | Version 1 | Code Corrected

CS-CSpend-007

The struct `TransferSpec` declares a field named `nonce` but different TransferSpec can use the same nonce and they are considered valid.

Code corrected:

The field `nonce` has been renamed to `salt`.

6.13 Missing Sanity Checks

Informational | Version 1 | Code Corrected

CS-CSpend-008

1. The function `GatewayMinter.initialize()` could check that input arrays `supportedTokens_` and `tokenMintAuthorities_` have the same length to avoid accidental misconfigurations.

Version 2:

2. The function `Deposits.depositFor()` could check that `depositor` is not address zero to prevent accidental deposits on behalf of the zero address in the wallet contract.

Code corrected:

The sanity check described in point 1 has been added in function `GatewayMinter.initialize()`. Similarly, the function `Deposits.depositFor()` now checks that `depositor` is non-zero.

6.14 Unused Functions

Informational Version 1 Code Corrected

CS-CSpend-011

The internal functions `getMetadata()` and `_asTransferSpec()` in `TransferSpecLib` are not used in the current version of the codebase.

Code corrected:

The unused functions have been removed in Version 2.

6.15 Validation of TypedMemView References

Informational Version 1 Specification Changed

CS-CSpend-012

The function `TransferSpecLib.getMetadata()` does not check the return value of `TypedMemView.slice()` which potentially can be `NULL`.

Specification changed:

The function `getMetadata()` has been removed from the codebase.

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 Non-canonical Conversion of Bytes to Address

Informational **Version 1**

CS-CSpend-009

The function `AddressLib._bytes32ToAddress()` implements the following statement to perform the type conversion:

```
function _bytes32ToAddress(bytes32 buf) internal pure returns (address) {
    return address(uint160(uint256(buf)));
}
```

Note that due to downcasting, higher bits of `buf` will be omitted. Thus, it is possible to have different input values `buf` that map to the same address.

7.2 Possible Gas Optimizations

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

CS-CSpend-010

We provide a non-exhaustive list of optimizations to make the codebase more efficient in terms of gas:

1. The mappings `denylistMapping`, `supportedTokens` and `usedHashes` use a type `bool` for the value. Solidity uses a word (256 bits) for each stored value and performs some additional operations when operating on `bool` values (due to masking). Therefore, using `uint` instead of `bool` is slightly more efficient.
2. The function `gatewayMint()` calls redundantly `getTransferSpec()` in function `_validateMintAuthorization()` and when passing the input to `_mintOrTransfer()`.
3. The argument `ids` in function `balanceOfBatch()` can be stored in calldata instead of memory.
4. The function `_moveBalanceToWithdrawing()` performs redundant SLOADs when returning values.

Version 2:

5. The constant variable `_CACHED_DOMAIN_SEPARATOR` does not get evaluated at compile time and included in the bytecode. Instead it points to a constant function that gets executed at runtime.
6. The function `TransferSpecLib.getTypedDataHash()` uses the identity precompile to copy data into memory, however using the `MCOPY` opcode is more efficient.

Code partially corrected:

The optimizations described in points 2, 3, and 4 have been implemented in **Version 2**.

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 Denylist on GatewayWallet and GatewayMinter

Note Version 1

Both contracts GatewayWallet and GatewayMinter keep a separate list of denylisted accounts that is independent of the denylist at the token level. The GatewayWallet prevents denylisted accounts from depositing tokens into the contract, updating delegations, or bridging (including same-chain transfers) tokens. However, denylisted users can still withdraw their tokens from the wallet contract.

8.2 Expanded Validity Window for Attestations

Note Version 1

The validity of an Attestation is based on the `block.number` of the destination chain. As a result, if block production on the destination chain slows down immediately after the Attestation is issued, the effective validity window of the Attestation becomes longer. In some cases, this window may even exceed that of the BurnIntent, especially if the destination chain experiences unexpectedly slow block production.

In general, `block.number` is not reliable for short time intervals (e.g., minutes) due to the possibility of missing or delayed blocks.

8.3 Expected Backend Checks

Note Version 1

Generally, we assume all checks performed on-chain on the `gatewayBurn()` call-path are validated first by the backend to ensure a BurnIntent always executes successfully on-chain.

Additionally, we assume the following checks are performed by backend before an Attestation is issued:

- The field "hook data length" in TransferSpec stores the correct length of the hook data.
- The length of the hook data in TransferSpec is restricted to ensure the execution of BurnIntent does not consume large amounts of gas, potentially making transactions revert on-chain due to gas limitations.
- The earliest expiry of a BurnIntent is no shorter than the withdrawal delay period.
- The user-provided signature for a BurnIntent is valid and the signer is authorized on-chain as delegatee of the depositor.
- The value specified in BurnIntent (sum of values in a BurnIntentSet) is less or equal to the available balance of the depositor.
- The field "max fee" is valid.
- The source token on the source domain matches the destination token on the destination domain.
- The source tokens for the same source domain in a BurnIntentSet should be the same, otherwise `gatewayBurn()` does not execute successfully.

- The actual fee charged for a BurnIntent is not extremely large even if the user specifies a large max fee (2**256-1), otherwise *gatewayBurn()* reverts due to overflows.
- The TransferSpec in an Attestation matches the respective TransferSpec in a BurnIntent.

8.4 Gas Cost Depend on Memory Expansion in gatewayBurn

Note **Version 1**

The function `gatewayBurn()` takes as input an array of burn intents, potentially from different users, each of which may include a set of BurnIntent entries signed by a single user. Although only a subset of all intents might be relevant for the local chain, all of them are copied into memory. We would like to highlight that the cost of memory expansion grows quadratically on the EVM; therefore, larger sets of BurnIntent consume more gas.

The off-chain service (attestation signers and BurnIntent signers) should take into account the gas cost of executing a BurnIntent on-chain. Large sets of BurnIntent may potentially be non-executable on-chain due to gas limits.

8.5 Implications of Chain Re-Organizations for Burn Intents

Note **Version 1**

The protocol design requires users to trust the off-chain service (the attestation signer and the burn signer) to perform token burns only after a valid Attestation has been executed on the destination chain. The off-chain service is expected to ensure that this assumption holds also in the presence of chain re-orgs.

If a re-org occurs on the destination chain, a previously valid Attestation may become invalid—particularly if its validity window is short. In such cases, the corresponding BurnIntent must not be executed on the source chain, as the burn condition is no longer satisfied.

Re-orgs on the source chain can similarly affect BurnIntent execution. To mitigate this risk, the `withdrawalDelay` parameter—configured by the owner—must be long enough to accommodate potential re-orgs on both chains involved in a TransferSpec.

8.6 Signatures From Revoked Delegatees Are Accepted by GatewayWallet

Note **Version 1**

The call-path for `gatewayBurn` checks if a BurnIntent is signed by an address that is authorized by the depositor. The check is implemented in the internal function `_wasEverAuthorizedForBalance()` which accepts signatures also from a revoked delegatee:

```
function _wasEverAuthorizedForBalance(...) ... returns (bool)
{
    // A depositor is always authorized for its own balance
    if (addr == depositor) return true;
    // Otherwise, check that the stored authorization status is either
    // `Authorized` or `Revoked`
    AuthorizationStatus status = DelegationStorage.get().
        authorizedDelegates[token][depositor][addr];
    return status != AuthorizationStatus.Unauthorized;
}
```

Users should trust the off-chain service (attestation signers) to monitor the state on chain and reject signatures from a revoked delegatee.

8.7 Total Supply of Tokens Onchain Temporarily Increases During Gateway Transfers

Note **Version 1**

This crosschain primitive is designed to first mint the trasferred tokens on the destination domain and then burn them later on the source chain. As a side effect, instant-transfers temporarily increase the total supply of the underlying token across all chains, until all pending burn intents are processed.

The total supply of the underlying tokens increases also for same-chain transfers as they work similar to cross-chain transfers since **Version 2**.

Circle business comment:

Circle is aware of this behavior, and ensures that reserve integrity is maintained throughout the Gateway transfer process.