



Circle Gateway Batching

Security Assessment

June 10th, 2026 — Prepared by OtterSec

Nicholas R. Putra

nicholas@osec.io

Zhenghang Xiao

kiprey@osec.io

Robert Chen

r@osec.io

Table of Contents

Executive Summary	2
Overview	2
Key Findings	2
Scope	3
General Findings	4
OS-CGB-SUG-00 Batching Denylist Check	5
OS-CGB-SUG-01 Batch Processing Order Assumption	6
OS-CGB-SUG-02 Code Maturity	7
OS-CGB-SUG-03 Gas Optimization	8
OS-CGB-SUG-04 Missing Validation Logic	9
Appendices	
Vulnerability Rating Scale	11
Procedure	12

01 — Executive Summary

Overview

Circle engaged OtterSec to assess the `gateway-batching` and `gateway-contract-signature` programs. This assessment was conducted between January 5th and January 8th, 2026. A follow-up review was performed between January 16th and January 21st, 2026. An additional follow-up review was performed between June 1st and June 2nd, 2026. For more information on our auditing methodology, refer to [Appendix B](#).

Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we made recommendations to add denylist checks for batching depositors ([OS-CGB-SUG-00](#)), enhance code maturity ([OS-CGB-SUG-02](#)), and apply targeted gas optimizations ([OS-CGB-SUG-03](#)). Lastly, we advised adding validation logic to prevent redundant operations and misleading events ([OS-CGB-SUG-04](#)).

02 — Scope

The source code was delivered to us in a Git repository at <https://github.com/circlefin/evm-gateway-contracts>. This audit was performed against [fd51093](#).

The final audit commit was [fd51093](#).

A brief description of the program is as follows:

Name	Description
gateway-batching	It adds support for the Circle Gateway, enabling an off-chain service to batch multiple transactions into a single on-chain transaction.
gateway-contract-signature	The update adds a Circle-managed TEE contractSignatureSigner path so smart-contract users can authorize Gateway burns via a TEE-produced ECDSA signature, while Gateway still treats the customer contract as the effective sourceSigner. It also adds owner-managed signer registration, splits allowlist/disallowlist roles for contract signers, updates wallet initialization, and disables accidental renounceOwnership() in GatewayWallet and GatewayMinter.

03 — General Findings

Here, we present a discussion of general findings identified during our audit. While these findings do not pose an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-CGB-SUG-00	The current batching implementation does not check whether the depositors involved in the transfer are on the denylist.
OS-CGB-SUG-01	The batching design assumes that off-chain submitted batches are processed strictly in order.
OS-CGB-SUG-02	Suggestions to improve consistency, safety, and adherence to best practices across the codebase.
OS-CGB-SUG-03	The overall efficiency of the codebase may be improved by including certain gas optimizations.
OS-CGB-SUG-04	Recommendations for adding validation logic to prevent redundant operations and misleading events.

Batching Denylist Check

OS-CGB-SUG-00

Description

In the current implementation of `submitBatch`, the code does not yet check whether the involved depositors are on the stored denylist, while other operations currently enforce this denylist check.

Remediation

Add a denylist check in `submitBatch` for each `depositor`.

Patch

Acknowledged by the Circle development team.

Batch Processing Order Assumption

OS-CGB-SUG-01

Description

Batch operations are designed for transfers and allow execution to proceed even if the **depositor** has insufficient balance, with an event emitted in such cases. Because underfunded execution is permitted, batch semantics are order-dependent and non-commutative.

Remediation

Ensure off-chain batch processing preserves strict ordering.

Patch

Acknowledged by the Circle development team.

Code Maturity

OS-CGB-SUG-02

Description

1. In `submitBatch`, the signatures lack an explicit expiration (deadline).
2. Adopt EIP-712 typed-data signing to strongly bind each batching signature and reduce future replay risk as new message formats and flows are introduced.
3. Add a per-depositor authorization signature for each batch, ensuring that even if the `batchSigner` is compromised, no funds can be moved without the depositor's permission.

Remediation

Implement the above-mentioned suggestions.

Patch

Acknowledged by the Circle development team.

Gas Optimization

OS-CGB-SUG-03

Description

1. Optimize `_checkAndMarkBatchIdUsed` by replacing `bytes32` with a monotonically increasing `uint256` and tracking usage with a bitmap.
2. Inline the `submitBatch` logic for balance increases and reductions, and wrap the deduction logic in `unchecked` since underflow is prevented by the existing if-else guard.

Remediation

Incorporate the suggestions above.

Patch

Acknowledged by the Circle development team.

Missing Validation Logic

OS-CGB-SUG-04

Description

1. Add an idempotency guard in `addContractSignatureSigner` and `removeContractSignatureSigner` in `ContractSignatureSigners`, so that `addContractSignatureSigner` reverts when `contractSignatureSigners[signer]` is already true, and `removeContractSignatureSigner` reverts when it is already false.

```
>_ src/modules/wallet/ContractSignatureSigners.sol SOLIDITY

function addContractSignatureSigner(address signer) public onlyOwner {
    AddressLib._checkNotZeroAddress(signer);
    ContractSignatureSignersStorage.get().contractSignatureSigners[signer] = true;
    emit ContractSignatureSignerAdded(signer);
}

function removeContractSignatureSigner(address signer) public onlyOwner {
    AddressLib._checkNotZeroAddress(signer);
    ContractSignatureSignersStorage.get().contractSignatureSigners[signer] = false;
    emit ContractSignatureSignerRemoved(signer);
}
```

2. Add a precondition in `ContractSignersAllowlist::disallowContractSigner` that requires `allowlistMapping[contractAddr] == ContractSignerAllowlistStatus.Allowlisted` before setting it to `Revoked`. This prevents revoking contracts that were never allowlisted or already revoked, and avoids misleading `ContractSignerDisallowed` events.

```
>_ src/modules/wallet/ContractSignatureSigners.sol SOLIDITY

function disallowContractSigner(address contractAddr) external
    ↪ onlyContractSignersDisallowlister {
    _setContractSignerAllowlist(contractAddr, ContractSignerAllowlistStatus.Revoked);
    emit ContractSignerDisallowed(contractAddr);
}
```

Remediation

Add the validations stated above.

Patch

1. Issue #1 was acknowledged by the Circle development team.
2. Issue #2 was resolved in [fd51093](#).

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the [General Findings](#).

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
 - Improperly designed economic incentives leading to loss of funds.
-

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
 - Exploitation involving high capital requirement with respect to payout.
-

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
 - Forced exceptions in the normal user flow.
-

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.
-

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
 - Improved input validation.
-

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.