Circle Gateway

Security Assessment

May 31st, 2025 — Prepared by OtterSec

Nicholas R. Putra	nicholas@osec.io
Michael Debono	mixy1@osec.io
Zhenghang Xiao	kiprey@osec.io
Robert Chen	r@osec.io

Table of Contents

01 — Executive Summary

Overview

Circle Finance engaged OtterSec to assess the `circle-gateway` contracts. This assessment was conducted between May 12th and May 24th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

In particular, we highlighted the possibility of front-running a user's deposit by utilizing their permit signature, resulting in the user's transaction to fail (OS-CGT-SUG-00). We also made suggestions regarding inconsistencies in the code base and ensuring adherence to coding best practices. (OS-CGT-SUG-01).

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 commit f24a7d1. Follow-up review was performed against commit ca774f1 and 5b5446f.

A brief description of the program is as follows:

Name	Description
circle-gateway	allows users to instantly mint USDC on any supported chain after depositing it on a source chain, using an off-chain authorization system.

02 — General Findings

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

ID	Description
OS-CGT-SUG-00	It is possible to front-run a user's `deposit` by utilizing their `permit` signature first, resulting in the user's transaction to fail.
OS-CGT-SUG-01	Suggestion regarding inconsistencies in the code base and ensuring adherence to coding best practices.

Deposit Failure due to Permit Front-Running OS-CGT-SUG-00

Description

If a user calls `permit` to approve token transfer, then follows it with `deposit`, an attacker may front-run the `deposit` utilizing the same `permit` signature. Since `permit` grants immediate approval, anyone may utilize it before the intended user's transaction is mined. Consequently, when the user's own `deposit` executes, it will fail due to insufficient allowance/balance.

Patch

`Circle` has acknowledged this as an acceptable risk, given that no effective remediation is currently feasible. Since the contract cannot distinguish between a utilized and an invalid permit, it requires a valid permit for deposits. Falling back to the existing allowances will introduce more severe risks, such as unauthorized fund transfers from users who have previously approved `GatewayWallet` for their `USDC`.

Code Maturity OS-CGT-SUG-01

Description

1. `Denylist::unDenylist`, the documentation incorrectly states it may only be called by the `owner`, while the code enforces access via the `denylister` role. This mismatch is misleading. The docstring should be corrected to reflect the actual access control and maintain consistency between implementation and specification.

   >_ *src/modules/common/Denylist.sol* SOLIDITY

   ```
   /// Allows a previously-denylisted address to interact with the contract again
   ///
   /// @dev May only be called by the `owner` role
   ///
   /// @param addr   The address to be allowed
   function unDenylist(address addr) external onlyDenylister {
       _denylist(addr, false);
       emit UnDenylisted(addr);
   }
   ```

2. The signature length check in `_verifyBurnerSignature` is redundant and may be removed since `ECDSA.recover` already checks this.
3. In `depositFor`, the documentation incorrectly states it transfers funds from the `depositor`, while the code actually transfers funds from the `sender`.

Remediation

Implement the above-mentioned suggestions.

Patch

The first and second issues have been resolved in ca774f1, and the third is acknowledged.

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.