OpenZeppelin | security

Uniswap ERC20ETH Audit

June 24, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-06-16 To 2025-06-18
Languages	Solidity

Total Issues	6 (6 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	2 (2 resolved)
Notes & Additional Information	4 (4 resolved)

Scope

OpenZeppelin audited the Uniswap/ERC20-eth repository at commit 65087ac. As additional resources, the Uniswap team provided the audit team with tests and integration examples for Uniswap's Calibur to support EIP-7702 and Permit2 functionality in pull request #227 and UniswapX settlement contracts in pull request #326.

In scope was the following file:

```
src
└── ERC20Eth.sol
```

System Overview

The draft ERC-7914 standard introduces a way for approved spenders to pull native tokens from smart contract wallets or EIP-7702 delegated EOA accounts. For those wallets that are capable of executing code, arbitrary contracts can be authorized to pull native tokens from them via a new callback mechanism.

The `ERC20ETH` contract leverages ERC-7914 to provide standard ERC-20 interfaces like `transfer`, `approve`, and `transferFrom`, while managing native token balances externally. The contract inherits from Solady's ERC-20 implementation, which inherently supports EIP-2612 Permit Extension and provides infinite allowance to Uniswap's Permit2 contract.

All transferring functions have been overridden and simplified to pull native tokens (such as ETH on mainnet) from the sender via ERC-7914's `transferFromNative` function call on the sender and immediately transfer them to the intended recipient. The sender can also transfer native tokens directly via the `transfer` function. Internal balance tracking has been eliminated to prevent double-entry-point balance check vulnerabilities.

The contract employs a minimal wrapper pattern to facilitate native token use within ERC-20 compatible ecosystems. It also allows smart contracts to adjust allowances for spenders via the standard approve functionality similar to ERC-20, and via the approve functionality of the `Permit2` contract as it has an infinite allowance.

The `permit` functionality in the contract only supports EOAs and ERC-7702 delegated EOAs while the signature validation method for smart accounts through the EIP-1271 standard is not supported. In order to support ERC20ETH token, the ERC-7914-compatible smart account must increase the allowance of the `ERC20ETH` contract by calling `approveFromNative` function on itself.

Security Model and Trust Assumptions

During the audit, the following trust assumptions were made:

- The ERC20ETH token is not fully ERC-20-compliant and should not be treated as such. It is a minimal wrapper to support native token transfers from ERC-7914-compatible smart accounts without the extra step of wrapping and/or unwrapping into/from WETH. As such, integrating ERC20ETH as an ERC-20 token can lead to major integration issues due to the following reasons:
  - The `totalSupply` function will always return 0.
  - The `balanceOf` function will always revert as there is no balance tracking mechanism in the contract.
  - The `transfer` and `transferFrom` functions have callback mechanism to the sender via `transferFromNative`.
  - The `recipient` must be an EOA or a smart contract that can handle incoming ETH via `receive` or `fallback` functionality.
- The `ERC20ETH` contract is not supposed to hold any native tokens. Any native tokens sent to this contract can be immediately transferred by anybody. This contract only temporarily pulls native tokens from the sender and immediately sends them to the recipient.
- ERC-7914-compatible smart accounts are assumed to send the exact amount of native tokens that was requested during the `transferFromNative` callback. If a greater amount is sent, the excess will remain in the contract and will be subsequently swept by any capable third party. For the same reason, no actor should ever send native tokens directly to this contract. A refund mechanism of any remaining amount might be an interesting alternative.
- When pulling native tokens from an account, the contract verifies that the balance of the contract matches the expected amount to transfer. However, it does not verify the source of the funds, which can lead to `Transfer` event spoofing.

Security Considerations for Integrators

The `ERC20ETH` contract essentially adds a valid ERC-20 address to represent the native token of the blockchain that it is deployed on. This creates a double entry point when referring to native tokens since many protocols use `address(0)` or `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE` to indicate an operation that deals with the native currency. A double entry point can turn into a double-spend vulnerability if not implemented correctly.

In addition, the `transferFromNative` callback into the sender's address creates yet another re-entry or gas griefing opportunity which may not have been envisioned when designing the protocol before EIP-7702 was activated. Similarly, recipients can execute fallback mechanisms upon receiving native tokens, giving them the ability to influence transaction flow. The off-chain relayers should thoroughly take gas griefing scenarios into consideration while supporting this token through the `permit` functionality or the `Permit2` contract.

It is strongly advised that any integrator looking to support the ERC20ETH token in their ecosystem carefully thinks about the implications of doing so and adapts their codebase accordingly.

Low Severity

L-01 Floating Pragma

Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled.

The `ERC20Eth.sol` file has the `solidity ^0.8.13` floating pragma directive.

Consider using fixed pragma directives for deployment. Keep in mind that starting with Solidity version 0.8.20, the `PUSH0` opcode was introduced, which is still unsupported by various chains at the time of writing.

Update: *Resolved in pull request #2.*

L-02 Incomplete Docstrings

Within the `ERC20ETH` contract, multiple instances of incomplete docstrings were identified:

- The `name`, `totalSupply`, and `symbol` functions are missing the `@return` tags to document their respective return values.
- Within the `balanceOf` function, the unnamed `address` parameter is not documented. However, this is reasonable because it is unused due to the function reverting unconditionally.
- Within the `totalSupply` function, there is a blank line between the function definition and its associated docstrings. This will lead to parsing tools not being able to identify them.

Consider thoroughly documenting all functions and events (and their parameters or return values) that are part of a contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #2.*

Notes & Additional Information

N-01 File and Contract Names Mismatch

The `ERC20Eth.sol` file name does not match the `ERC20ETH` contract name.

To make the codebase easier to understand for developers and reviewers, consider renaming the files to match the contract names.

Update: *Resolved in pull request #4.*

N-02 Missing Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

The `ERC20ETH` contract does not have a security contact.

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Resolved in pull request #2.*

N-03 Inconsistency in Named Return Variables

Named return variables are a way to declare variables that are meant to be used within a function's body for the purpose of being returned as that function's output. They are an alternative to explicit in-line return statements.

Throughout the codebase, named return variables are not defined except for one instance which is also unused. For instance, in `ERC20Eth.sol`, the `result` return variable of the `totalSupply` function is unused.

Consider removing the aforementioned named return variable to ensure consistency with the design decision implemented throughout the contract for hardcoded return values.

Update: *Resolved in pull request #2.*

N-04 Unlicensed Code

The use of `// SPDX-License-Identifier: UNLICENSED` in the code indicates that no explicit license has been specified. This has significant implications, such as:

- **Restricted Usage by Default**: Without a license, others are not granted permission to copy, modify, or distribute the code. Copyright laws apply by default, protecting the code, and any use without permission could be considered copyright infringement.
- **Enforceability Concerns**: Even though the code is legally protected, the absence of a clear license might make it harder to enforce legal rights. Some users may mistakenly believe that the code is open for use due to the lack of a license statement.
- **No Attribution Requirements**: Since no license is provided, there are no usage conditions such as attribution requirements. However, others are not permitted to use the code at all without explicit permission from the author.

To ensure control over how the code is used, consider specifying a license. A license provides legal protection by establishing clear usage terms which helps prevent misuse and encourages collaboration.

Update: *Resolved in pull request #3.*

Conclusion

This audit focused on the `ERC20ETH` contract and the potential impact of integrating it into the Uniswap ecosystem. Only minor issues were identified, and several considerations were raised around the risks of incorrectly integrating the ERC20ETH token with other DeFi contracts/systems. The Uniswap team is appreciated for being highly cooperative and providing clear documentation and communication throughout the audit period.