Ultra Bridge ETH Audit

Prepared on: 2026-01-01

Contract: AUD534

Prepared by:
charles Holtzkampf

Prepared for:
ultra

Table of Contents

Executive Summary

This document outlines any issues found during the audit of the contracts:

- ultra-bridge-eth
- fbee7f22d04c4ac0e55db5b578c62d18c50914d1

- The contract has 2 remarks.
- 7 Minor security issues were found.
- 2 Major security issues were found.
- 0 Critical security issues were found.
- The risk associated with this contract is low

REMARK	MINOR	MAJOR	CRITICAL
2	7	2	0

Severity Description

REMARK

Remarks are instances in the code that are worthy of attention, but in no way represent a security flaw in the code. These issues might cause problems with the user experience, confusion with new developers working on the project, or other inconveniences.
Things that would fall under remarks would include:

- Instances where best practices are not followed
- Spelling and grammar mistakes
- Inconsistencies in the code styling and structure

MINOR

Issues of Minor severity can cause problems in the code, but would not cause the code to crash unexpectedly or for funds to be lost. It might cause results that would be unexpected by users, or minor disruptions in operations. Minor problems are prone to become major problems if not addressed appropriately.
Things that would fall under minor would include:

- Logic flaws (excluding those that cause crashes or loss of funds)
- Code duplication
- Ambiguous code

MAJOR

Issues of major security can cause the code to crash unexpectedly, or lead to deadlock situations.
Things that would fall under major would include:

- Logic flaws that cause crashes
- Timeout exceptions
- Incorrect ABI file generation
- Unrestricted resource usage (for example, users can lock all RAM on contract)

CRITICAL

Critical issues cause a loss of funds or severely impact contract usage.
Things that would fall under critical would include:

- Missing checks for authorization
- Logic flaws that cause loss of funds
- Logic flaws that impact economics of system
- All known exploits (for example, on_notification fake transfer exploit)

Methodology

Throughout the review process, we check that the token contract:

- Documentation and code comments match logic and behaviour
- Is not affected by any known vulnerabilities

Our team follows best practices and industry-standard techniques to verify the proper implementation of the smart contract. Our smart contract developers reviewed the contract line by line, documenting any issues as they were discovered. Ontop of the line by line review, we also perform code fuzzzing.

Our strategies consist largely of manual collaboration between multiple team members at each stage of the review, including:

I. Due diligence in assessing the overall code quality of the codebase.
II. Testing contract logic against common and uncommon attack vectors.
III. Thorough, manual review of the codebase, line-by-line.

Our testing includes:

- Overflow Audit
- Authority Control
- Authority Vulnerability
- Re-entry
- Timeout
- RAM Attacks
- Fake contract
- Fake deposit
- Denial of Service
- Design Logic Audit
- RNG attacks
- Stack Injection attacks

Our Code Fuzzing methodology:

We use a modified **EOSIO Contract Development Kit (CDT)** and custom harnessing to make EOS contracts suitable for fuzzing. The contract is instantiated and it's public functions are called in random order, with random input, so as to explore the state space and find corner case inputs that might lead to undesired outcomes. Apart from detecting logic bugs, this approach allows us also to detect memory bugs, hangs, undefined behavior and crash bugs in a semi-automated manner. Since EOS contracts are usually designed to run and complete within a short amount of time, fuzzing them is very fast as well, and therefore an effective instrument for teasing out bugs within the duration of an audit.

HIGH SEVERITY

1 H1: Hardcoded Counter Gap Limit Enables Bridge DoS

Location: `UltraBridge.sol` → `claim2EVM()` →

Status: Resolved in commit: `6c8c17c00b2cea7445e08c4006db39a00222b82f`
Counter gap limit removed entirely. Replaced with O(1) `claimedUltra2EVMSwapCounters` mapping.

The 10,000 counter gap limit can be exploited to completely block the bridge:

- **Attacker spams swaps on Ultra** - creates 10,000 pending claims
- **Attacker never claims** - `highestClaimedUltra2EVMCounter` stays fixed
- **All legitimate claims blocked** - counter 10,001+ fails the gap check
- **Natural accumulation** - even without attack, unclaimed swaps over time hit the limit

```
require(data.counter < highestClaimedUltra2EVMCounter + 10000, "There are too many
↪  unclaimed swaps");
```

Resolution: Make the gap limit configurable to allow operational adjustment or remove entirely:

```
uint64 public maxCounterGap = 10000;

function setMaxCounterGap(uint64 newGap) external onlyOwner {
    maxCounterGap = newGap;
}
```

2 H2: O(n) Counter Loop Creates Expensive Claims and Gas DoS

Location: `UltraBridge.sol` → `_updateCounter()` → Lines 783-785

Status: Resolved in commit: `6c8c17c00b2cea7445e08c4006db39a00222b82f`
`_updateCounter()` loop removed. All claims now cost constant gas via O(1) claimed mapping.

`_updateCounter()` writes a storage slot for every unclaimed counter between `highestClaimedUltra2EVMCo`
and the claimed counter. This creates two problems:

- **Griefing attack** - Attacker submits 100+ swaps, making legitimate claims expensive
- **User cost escalation** - Later users in the queue pay more gas than earlier users
- **Block limit DoS** - Large gaps can exceed block gas limits, making claims impossible until the gap is closed

```
for (uint64 i = highestClaimedUltra2EVMCounter + 1; i < counter; i++) {
    unclaimedUltra2EVMSwapCounters[i] = true;  // ~20,000 gas per write
}
```

Resolution: Replace O(n) gap-fill with O(1) claimed mapping:

```
mapping(uint64 => bool) public isClaimed;
require(!isClaimed[data.counter], "Already claimed");
isClaimed[data.counter] = true;
```

This makes all claims cost the same regardless of queue position.

LOW SEVERITY

1 L1: Missing Overflow Check When EVM and Ultra Decimals Match

Location: `UltraBridge.sol` → `setSupportedToken()`

Status: Resolved in commit: `6c8c17c00b2cea7445e08c4006db39a00222b82f`
Added `else` branch with `if (supportedToken.maxSwapAmount > MAX_ULTRA_AMOUNT) revert MaxSwapAmountOverflow();`

Overflow checks exist for when `evmDecimals > ultraPrecision` and `evmDecimals < ultraPrecision`, but no check exists when they are equal. If owner sets `maxSwapAmount > MAX_ULTRA_AMOUNT` and decimals match, swaps could exceed Ultra's representable range.

```
if(evmDecimals > supportedToken.ultraPrecision){
    // check exists
} else if (evmDecimals < supportedToken.ultraPrecision) {
    // check exists
}
// Missing: no check when evmDecimals == ultraPrecision
```

Resolution:

```
if(evmDecimals > supportedToken.ultraPrecision){
    // check exists
} else if (evmDecimals < supportedToken.ultraPrecision) {
    // check exists
}
else {
    // evmDecimals == ultraPrecision
    require(supportedToken.maxSwapAmount <= MAX_ULTRA_AMOUNT, "max swap amount exceeds
    ↪  ultra max amount");
}
```

2 L2: Fee-on-Transfer Token Policy Not Enforced

Location: `UltraBridge.sol` → `swap2Ultra()`

Status: Resolved in commit: `6c8c17c00b2cea7445e08c4006db39a00222b82f`

The code comments state "we don't support token with tax/burn/fee/etc" but this policy is not enforced. The check only validates that `received` is within min/max bounds, not that `received == bridgingAmount`.

```
IERC20(address(token)).safeTransferFrom(msg.sender, address(this), bridgingAmount);
uint256 received = IERC20(token).balanceOf(address(this)) - before;

require(
    received >= stoken.minSwapAmount &&
    received <= stoken.maxSwapAmount,  // Only checks range, not equality
    "Amount out of range after fee/dust"
);
```

A fee-on-transfer token could pass validation if the reduced amount is still within range.

Resolution:

```
require(received == bridgingAmount, "Fee-on-transfer tokens not supported");
```

INFORMATIONAL

1 I1: Debug Import Present

Location: `UltraBridge.sol` → Line 19

`import "hardhat/console.sol"` should be removed for production.

2 I2: Commented Code Sections Remain

Location: `UltraBridge.sol` → Lines 12, 569-584, 614

Dead/commented code should be removed before deployment.

3 I3: Magic Numbers Used

Location: Throughout `UltraBridge.sol`

Magic numbers (5000, 10000, version "1") should be named constants.

Resolution:

```
uint256 private constant GAS_RESERVE = 5000;
uint64 private constant MAX_COUNTER_GAP = 10000;
string public constant VERSION = "1.0.0";
```

4 I4: Dead Code in _findSchedule()

Location: `UltraBridge.sol` → `_findSchedule()` → Lines 751-754

The else branch is unreachable. `_findSchedule()` is only called with `block.timestamp` as the `timestamp` parameter, making the nested check redundant.

```
function _findSchedule(uint256 timestamp) private view returns (...) {
    for(uint64 i = scheduleVersion; i > 0; i--) {
        if(validatorSchedules[i].startTime <= timestamp) {
            if(validatorSchedules[i].startTime <= block.timestamp) {  // Redundant
                return (validatorSchedules[i], i);
            } else {
                // DEAD CODE
                return (ValidatorSchedule(new address[](0), 0, 0), 0);
            }
        }
    }
}
```

Resolution: Remove the redundant inner check and dead else branch.

5 I5: Missing Events for Configuration Changes

Location: `UltraBridge.sol` → `setSupportedToken()`, `setValidatorSchedule()`

These functions emit no events, reducing transparency for off-chain monitors.

Resolution:

```
event SupportedTokenUpdated(address indexed token, SupportedToken data);
event ValidatorScheduleUpdated(uint64 indexed version, address[] validators, uint64
↪  threshold, uint256 startTime);
```

6 I6: Quadratic Complexity in Validator Duplicate Check

Location: `UltraBridge.sol` → `setValidatorSchedule()` → Lines 457-462

O(n²) duplicate detection in validator uniqueness check scales poorly.

```
for (uint i = 0; i < validatorAddresses.length; i++) {
    for (uint j = i + 1; j < validatorAddresses.length; j++) {
        require(validatorAddresses[i] != validatorAddresses[j], "Duplicate
        ↪  validator");
    }
}
```

Resolution: Use mapping-based O(n) approach for duplicate detection if validator count grows significantly.

7 I7: Missing Zero Address Validation in BridgeToken

Location: `BridgeToken.sol` → `initialize()`

`BridgeToken.initialize()` doesn't validate `initialOwner_ != address(0)`.

Resolution:

```
require(initialOwner_ != address(0), "Invalid owner");
```