cecuro

Audit Report

March 2, 2026

PROJECT

Whitechain

Audit Overview

Project:	Whitechain
Repository:	https://github.com/whitechain-labs/bridge-contracts
Audit Date:	March 2, 2026
Commit:	a9b7a82e
Scope:	3 files

0	0	3	11	8
CRITICAL	HIGH	MEDIUM	LOW	INFO

Audit Scope

The following 3 files were included in this security audit:

```
ether/contracts/main/modules/bridge/Bridge.sol
```

```
ether/contracts/main/modules/mapper/Mapper.sol
```

```
ether/contracts/main/libraries/ECDSAChecks.sol
```

Findings

MEDIUM

Missing canonical validation for bytes32-to-address casts can burn funds and bypass per-token accounting/limits

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:267-305`

`ether/contracts/main/modules/bridge/Bridge.sol:281-285`

`ether/contracts/main/modules/bridge/Bridge.sol:298-304`

`ether/contracts/main/modules/bridge/Bridge.sol:529-610`

`ether/contracts/main/modules/mapper/Mapper.sol:118-169`

Description:

The system represents token identifiers and recipient addresses as `bytes32`, then later converts them to EVM `address` via `address(uint160(uint256(x)))`.

However, there is **no validation that the high 96 bits are zero** (canonical EVM encoding), and the only guard used is `nonZeroBytes32(bytes32)` which does **not** prevent the derived EVM address from being `address(0)`.

This creates multiple issues: 1) **Fund burn / misdirection**: In `receiveTokens()`, a non-zero `bytes32` `toAddress` with low 20 bytes = 0 will pass `nonZeroBytes32`, but the casted address becomes `address(0)`, so ETH/ERC20 can be irreversibly sent to `address(0)`. 2) **Accounting/limit bypass & mapping collisions**: Token identifiers in Mapper are keyed by full `bytes32`, but Bridge truncates them to 160 bits when interacting with ERC20s. Different `bytes32` values that share the same low 20 bytes will be treated as the *same* ERC20 by Bridge but as *different* tokens by Mapper/dailyLimits/dailyVolumes, enabling configuration mistakes or deliberate bypass of per-token uniqueness and per-token daily limits.

Evidence:

`receiveTokens()` casts `toAddress` and uses it for value transfer:

```
payable(address(uint160(uint256(receiveTokensParams.toAddress)))).call{ value: receiveTo-
kensParams.amount }("");
...
_executeTokenTransfer(... to: address(uint160(uint256(receiveTokensParams.toAddress))), ...);
```

(Bridge.sol:281-285, 298-304)

Only `nonZeroBytes32(receiveTokensParams.toAddress)` is enforced (Bridge.sol:258-259), which does not guarantee the casted address is non-zero or canonical.

ERC20 addresses are also truncated in `_executeTokenTransfer`, `_executeToken-TransferFrom`, and `_getBalance` (Bridge.sol:529-610).

Mapper stores token identifiers as raw `bytes32` keys without canonical checks (Mapper.sol:118-169).

Impact:

- **Irrecoverable loss**: ETH/tokens can be sent to `address(0)` if `toAddress` is non-canonically encoded (operator error or malicious relayer).
- **Limit bypass / inconsistent state**: The same ERC20 may be registered multiple times under different `bytes32` identifiers, bypassing Mapper uniqueness checks and splitting `dailyLimits/dailyVolumes` by `bytes32` key.

Recommendation:

Enforce canonical encoding wherever a `bytes32` is expected to represent an EVM address:

- require `(uint256(x) >> 160) == 0` (high bits zero)
- require `address(uint160(uint256(x))) != address(0)` when applicable (recipient/token must not be zero)

Apply similar validation in `Mapper.registerMapping()` for EVM-chain deployments and in Bridge before using `toAddress`/token addresses for transfers.

MEDIUM

Fee-on-transfer/deflationary tokens can cause recipients to receive less than the bridged amount on Unlock withdrawals

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:251-317`

`ether/contracts/main/modules/bridge/Bridge.sol:529-567`

Description:

On deposits, the bridge computes `actualAmount` using `balanceAfter - balanceBefore`, which correctly handles fee-on-transfer tokens.

On withdrawals for `WithdrawType.Unlock` (token transfers), the bridge **blindly transfers `receiveTokensParams.amount`** and then emits `Withdrawal` for that same amount. For fee-on-transfer / deflationary tokens, the recipient will receive **less than `amount`**, while the bridge will still report that the full amount was withdrawn.

Evidence:

Deposit-side uses balance delta:

```
uint256 balanceBefore = _getBalance(_mapInfo.originTokenAddress);
_executeTokenTransferFrom(..., _amount);
uint256 balanceAfter = _getBalance(_mapInfo.originTokenAddress);
actualAmount = balanceAfter - balanceBefore;
```

Withdrawal-side does not:

```
_executeTokenTransfer(..., receiveTokensParams.amount);
emit Withdrawal(..., amount: receiveTokensParams.amount, ...);
```

Impact:

- Users may be systematically shortchanged on the receiving chain.
- Off-chain accounting/relayers indexing `Withdrawal.amount` will be incorrect.
- Bridge liquidity can be depleted faster than users receive value (transfer fees siphon value).

Recommendation:

Either (a) disallow fee-on-transfer / rebasing tokens at mapping time, or (b) measure actual received amount on withdrawal (balance delta of recipient or contract) and emit/use that amount consistently.

MEDIUM

Unsafe ERC20 transfer/transferFrom path (useTransfer=true) can silently fail and still emit events / consume limits

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:529-567`

`ether/contracts/main/modules/bridge/Bridge.sol:173-245`

`ether/contracts/main/modules/bridge/Bridge.sol:251-317`

> **Comment**
> The useTransfer flag is an opt-in configuration per mapping — token integrators choose the appropriate transfer method for their token. Mappings using safeTransfer (the default) are not affected.

Description:

When `useTransfer` is true, Bridge uses raw `IERC20Upgradeable.transfer()` / `transferFrom()` and **does not check the returned boolean**. For ERC20 tokens that return `false` instead of reverting (or otherwise behave non-standardly), Bridge will continue execution as if the transfer succeeded.

This creates inconsistent on-chain state and off-chain accounting:

- `receiveTokens()` updates the daily limit tracker and emits `Withdrawal` even if no tokens were actually transferred.
- `bridgeTokens()` may emit `Deposit` with `actualAmount = 0` after a failed `transferFrom`, while still consuming the relayer signature (`usedHashes`) and accumulating gas fees.

Phase-1 suspect `vp_cross_contract_setDailyLimit_d0b0db8c` points to this issue (actual location is `_executeTokenTransfer*`).

Vulnerable Code:

```
function _executeTokenTransfer(bool useTransfer, bytes32 tokenAddress, address to, uint256
amount) private {
 if (useTransfer) {
 IERC20Upgradeable(address(uint160(uint256(tokenAddress)))).transfer({ to: to, amount:
amount }); // return value ignored
 } else {
 IERC20Upgradeable(address(uint160(uint256(tokenAddress)))).safeTransfer({ to: to, value:
amount });
 }
}

function _executeTokenTransferFrom(bool useTransfer, bytes32 tokenAddress, address from,
address to, uint256 amount) private {
 if (useTransfer) {
 IERC20Upgradeable(address(uint160(uint256(tokenAddress)))).transferFrom({ from: from, to:
to, amount: amount }); // return value ignored
 } else {
 IERC20Upgradeable(address(uint160(uint256(tokenAddress)))).safeTransferFrom({ from: from,
to: to, value: amount });
 }
}
```

Impact:

- **User fund loss / stuck payouts**: `receiveTokens()` can emit a successful withdrawal event while transferring nothing.
- **Daily limit corruption / DoS**: daily volume can be incremented even if payout didn't happen, preventing future legitimate withdrawals until reset.
- **Operational incompatibility** with non-standard ERC20s: raw `transfer/transferFrom` may also revert for tokens that don't return a boolean (common legacy tokens), breaking bridging for those mappings.

Recommendation:

- Remove the raw `transfer/transferFrom` branch and always use `SafeERC20`.
- If you must keep it, require the return value is `true` and/or wrap it with low-level call logic equivalent to OZ `SafeERC20`.
- Add a sanity check like `require(actualAmount > 0)` for token deposits if `amount > 0` is expected.

LOW

receiveTokens has no on-chain replay protection (externalId not tracked) enabling double mint/unlock by a relayer

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:251-317`

`ether/contracts/main/modules/bridge/interfaces/IBridge.sol:62-68`

> **Comment**
> By design, the relayer is a permissioned role responsible for submitting each cross-chain message exactly once. Deduplication is enforced at the relayer layer, and daily limits serve as an additional safeguard against excessive outflows.

Description:

`Bridge.receiveTokens()` is the core withdrawal/mint/unlock entry point. Although it accepts an `externalId` (presumably the source-chain tx/message identifier), the contract **never stores or checks** that `externalId` (or any message hash) has been processed before.

As a result, any address with `RELAYER_ROLE` can call `receiveTokens()` multiple times with the same `externalId`/parameters (or even with different `externalId`s) and repeatedly mint/unlock assets, constrained only by the configured `dailyLimits`.

This is a classic bridge safety invariant violation: *each source-chain event/message should be consumable at most once on the destination chain*.

Evidence:

`externalId` exists in the interface:

```
struct ReceiveTokensParams {
 bytes32 externalId;
 uint256 mapId;
 uint256 amount;
 bytes32 fromAddress;
 bytes32 toAddress;
}
```

But in `receiveTokens` it is only emitted, never checked/stored:

```
function receiveTokens(ReceiveTokensParams calldata receiveTokensParams)
 external
 nonReentrant
 onlyRole(RELAYER_ROLE)
{
 ...
 _checkAndUpdateDailyLimit(tokenAddress, relayer, receiveTokensParams.amount);

 if (_mapInfo.isCoin) {
 ...
 payable(to).call{value: receiveTokensParams.amount}("");
 } else {
 if (_mapInfo.withdrawType == IMapper.WithdrawType.Mint) {
 IERC20Mintable(targetToken).mint(to, receiveTokensParams.amount);
 } else {
 _executeTokenTransfer(...);
 }
 }

 emit Withdrawal(..., receiveTokensParams.externalId, ...);
}
```

Impact:

- If a relayer key is compromised or the relayer software malfunctions, the attacker/bug can **double-spend** withdrawals: mint/unlock the same “bridged” amount multiple times.
- Even with daily limits, damage can be up to the full daily limit per token (and for native coin, `bytes32(0)` limit), repeated day after day.
- This breaks the bridge’s fundamental accounting and can drain liquidity (unlock path) or inflate supply (mint path).

Recommendation:

Add on-chain consumption tracking for withdrawals, e.g. `mapping(bytes32 => bool) processedWithdrawals;` keyed by `externalId` (or a stronger hash over all critical parameters including chain IDs, contract address, and `externalId`). Require it to be unused and mark it used *before* performing mint/unlock.

LOW

Balance-delta accounting (balanceAfter - balanceBefore) can be manipulated or underflow with rebasing/malicious tokens, leading to incorrect bridged amounts or DoS

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:212-234`

`ether/contracts/main/modules/bridge/Bridge.sol:414-435`

Description:

For ERC20 deposits, `Bridge.bridgeTokens()` and `Bridge.depositTokens()` compute the received amount as `balanceAfter - balanceBefore`.

This pattern is commonly used to support fee-on-transfer tokens, but it assumes that:

- `balanceOf(address(this))` is honest and stable during the call, and
- the token's `transferFrom` only affects balances by moving tokens from `from` to `to`.

For rebasing/reflect tokens, tokens with transfer hooks, or malicious ERC20s, the token can change the bridge's balance for reasons unrelated to the user's transfer (e.g., minting to the bridge during `transferFrom`, burning from the bridge during `transferFrom`, or otherwise manipulating `balanceOf`).

Two concrete failure modes: 1) **Inflated actualAmount**: token mints/transfers extra tokens to the bridge during `transferFrom` ' `actualAmount` becomes larger than the user-intended `amount`, and the `Deposit` event reports the larger value. 2) **Underflow/DoS**: token reduces the bridge's balance during the call such that `balanceAfter < balanceBefore` ' `balanceAfter - balanceBefore` underflows and reverts in Solidity 0.8+, permanently blocking bridging for that mapping.

Evidence:

```
uint256 balanceBefore = _getBalance({ tokenAddress: _mapInfo.originTokenAddress });
_executeTokenTransferFrom(...);
uint256 balanceAfter = _getBalance({ tokenAddress: _mapInfo.originTokenAddress });
actualAmount = balanceAfter - balanceBefore;
```

(Bridge.sol:214-228)

Same pattern in `depositTokens()` (Bridge.sol:415-429).

Impact:

- **Incorrect cross-chain accounting** (over/under-mint or over/under-unlock depending on the remote chain implementation).
- **Denial of service** for specific token mappings if underflow occurs.

Recommendation:

Explicitly restrict supported tokens (reject rebasing/reflect/malicious tokens) and/or use a safer accounting model depending on intended support:

- For standard tokens: require `actualAmount == amount`.
- For fee-on-transfer support: keep balance-delta but add explicit documentation and allowlisting, and consider additional invariants (e.g., upper bounds on `actualAmount`).

LOW

Bridge implementation contract can accept ETH via receive() but cannot withdraw it (funds can be permanently stuck on the implementation)

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:120-133`

`ether/contracts/main/modules/bridge/Bridge.sol:322-334`

`ether/contracts/main/modules/bridge/Bridge.sol:369-394`

Description:

`Bridge`'s implementation contract includes a payable `receive()` function that accepts ETH and emits `CoinsDeposited`. The constructor calls `_disableInitializers()`, so the implementation can never be initialized, and therefore can never have roles granted.

As a result, if anyone mistakenly sends ETH directly to the **implementation address** (instead of the proxy address), the ETH is accepted but becomes **unrecoverable**, because all ETH-withdrawal paths are role-gated and the implementation can never be granted those roles.

This is a common upgradeable-contract footgun: the implementation should ideally not accept funds (or should have an explicit recovery mechanism), since it is not the intended custody address.

Evidence:

Implementation accepts ETH:

```
constructor() {
 _disableInitializers();
}

receive() external payable {
 emit CoinsDeposited({ account: _msgSender(), amount: msg.value });
}
```

Withdrawals require roles that can never be granted on the implementation:

```
function withdrawGasAccumulated() external ... onlyRole(EMERGENCY_ROLE) ...

function withdrawCoinLiquidity(...) external ... onlyRole(MULTISIG_ROLE) ...
```

Impact:

Low impact / medium likelihood: users/operators can accidentally lock ETH forever by sending it to the implementation address (e.g., via block explorer UI confusion).

Recommendation:

Avoid accepting ETH on the implementation (e.g., remove/disable `receive()` on logic contracts), and/or add an explicit recovery pattern consistent with your security model.

LOW

Disabling/removing mappings can strand assets owed to users (no on-chain recovery path)

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:262-305`

`ether/contracts/main/modules/mapper/Mapper.sol:94-191`

Description:

`Bridge.receiveTokens()` requires `mapInfo.isAllowed == true` and that the `mapId` resolves to a valid mapping. `Mapper.disableMapping()` and `Mapper.removeMapping()` can make this impossible.

If the bridge operates in a lock/burn on chain A ’ unlock/mint on chain B model, then after users have already locked/burned assets on the origin chain, disabling/removing the corresponding withdrawal mapping on the destination chain can prevent release of the owed assets.

Evidence:

Bridge blocks withdrawals on disabled mappings:

```
IMapper.MapInfo memory _mapInfo = _getMapInfo({ mapId: receiveTokensParams.mapId });
require(_mapInfo.isAllowed, "Bridge: IsAllowed must be true");
```

Mapper can disable/remove mappings:

```
function disableMapping(uint256 mapId) external onlyRole(EMERGENCY_ROLE) {
 mapInfo[mapId].isAllowed = false;
}

function removeMapping(uint256 mapId) external onlyRole(EMERGENCY_ROLE) {
 delete mapInfo[mapId];
}
```

Impact:

- In emergencies or operational mistakes, users can be left with funds locked/burned on the origin chain while the destination chain cannot complete the corresponding outflow.

- Recovery depends entirely on privileged actors re-enabling/recreating mappings; users have no on-chain claim path.

Recommendation:

Consider separating “pause deposits” from “pause withdrawals”, or providing a controlled recovery flow that can still honor already-observed inbound transfers while preventing new deposits.

LOW

Mapped token contracts can make bridge operations unexecutable via gas/compute blowups (no constraints or recovery), potentially stranding funds

Locations:

```
ether/contracts/main/modules/bridge/Bridge.sol:214-234
ether/contracts/main/modules/bridge/Bridge.sol:292-304
ether/contracts/main/modules/bridge/Bridge.sol:404-435
ether/contracts/main/modules/bridge/Bridge.sol:529-567
ether/contracts/main/modules/bridge/Bridge.sol:608-610
```

Description:

Bridge execution relies on calling arbitrary, mapped token contracts for core state transitions:

- Deposits: `balanceOf` (twice) + `transferFrom` + optional `burn`.
- Withdrawals: optional `mint` or `transfer`/`safeTransfer`.

Neither `Bridge` nor `Mapper` constrains the complexity of these external token calls. If a mapped token implementation is unusually expensive (e.g., `balanceOf`/`transfer-`/`mint`/`burn` performs heavy computation, large storage iteration, or returns pathological return data), the required gas can exceed the block gas limit. In that case, **deposits and/or withdrawals for that mapping become permanently unexecutable** on-chain.

Because withdrawals are push-based (relayer calls `receiveTokens()`), there is no on-chain "claim/pull" alternative for users to complete withdrawals if the token transfer path is too expensive.

This is an execution-flow / computational-bounds risk driven by configuration (which tokens are mapped).

Evidence:

Token deposits compute "actualAmount" by doing two external `balanceOf` calls around a transfer:

```
uint256 balanceBefore = _getBalance({ tokenAddress: _mapInfo.originTokenAddress });
_executeTokenTransferFrom(...);
uint256 balanceAfter = _getBalance({ tokenAddress: _mapInfo.originTokenAddress });
actualAmount = balanceAfter - balanceBefore;
```

(Bridge.sol:214-228)

Withdrawals depend on external token behavior (`mint` or `transfer`/`safeTransfer`):

```
if (_mapInfo.withdrawType == IMapper.WithdrawType.Mint) {
 IERC20Mintable(...).mint(...);
} else {
 _executeTokenTransfer(...);
}
```

(Bridge.sol:292-304)

Impact:

- A misconfigured or malicious token mapping can cause a **permanent DoS** for deposits/withdrawals for that token on this chain.
- If funds are locked in the bridge for that token (Lock/Unlock model), users' assets can be **stranded** without a user-facing recovery mechanism.

Recommendation:

- Treat token mapping as a strict allowlist with explicit operational requirements (standard ERC-20 behavior, bounded gas).
- Consider reducing external call count (e.g., avoid double-`balanceOf` where possible) or adding alternative recovery/claim flows for withdrawals that cannot be executed via direct token transfer.

LOW

Mapper allows enabling/disabling non-existent mapping IDs (e.g., mapId=0 or removed entries)

Locations:

`ether/contracts/main/modules/mapper/Mapper.sol:94-112`

`ether/contracts/main/modules/mapper/Mapper.sol:175-191`

Description:

`Mapper.enableMapping()` / `disableMapping()` only check `mapCounter >= mapId` to validate existence. This condition allows:

- `mapId == 0` (always passes), even though valid mappings start at 1.
- `mapId` that was previously deleted via `removeMapping()` (the `mapInfo[mapId]` struct becomes all-zero again).

These functions can then flip `mapInfo[mapId].isAllowed` for entries that have no associated metadata.

Evidence:

```
function enableMapping(uint256 mapId) external onlyRole(MULTISIG_ROLE) {
 require(mapCounter >= mapId, "Mapper: MapCounter must be greater than or equal mapId");
 require(!mapInfo[mapId].isAllowed, "Mapper: IsAllowed must be false");
 mapInfo[mapId].isAllowed = true;
}
```

No check that `mapId != 0` or that the mapping entry is populated.

Impact:

While Bridge entrypoints additionally check mapping fields (depositType/withdrawType/token addresses) such that a blank mapping typically cannot be used for a successful bridge, this is still a correctness issue that can:

- Confuse operators and indexers (enabled mapping that is not real).
- Increase risk of admin mistakes during emergency response/remediation.

Recommendation:

Tighten validation (e.g., `require(mapId != 0 && mapId <= mapCounter)` and/or require a non-zero sentinel field like `originChainId != 0` before enabling/disabling).

LOW

Mapper.initialize lacks zero-address validation; misconfiguration can permanently brick admin/upgrade controls

Location:

`ether/contracts/main/modules/mapper/Mapper.sol:81-88`

Description:

`Mapper.initialize()` grants `DEFAULT_ADMIN_ROLE` and `MULTISIG_ROLE` to `initParams.multisigAddress` and grants `EMERGENCY_ROLE` to `initParams.emergencyAddress`, but it does **not** validate these addresses are non-zero.

If initialization is performed with `address(0)` for either role (accidentally or via bad deployment tooling), the contract can become permanently stuck:

- If `multisigAddress == address(0)`, no one can call `enableMapping`, `registerMapping`, or authorize UUPS upgrades.
- If `emergencyAddress == address(0)`, no one can call `disableMapping`/`removeMapping` in emergencies.

Unlike Bridge.initialize, Mapper.initialize has no `nonZeroAddress` checks.

Evidence:

```
function initialize(InitParams calldata initParams) external initializer {
 __UUPSUpgradeable_init();
 __AccessControl_init();

 _grantRole(DEFAULT_ADMIN_ROLE, initParams.multisigAddress);
 _grantRole(MULTISIG_ROLE, initParams.multisigAddress);
 _grantRole(EMERGENCY_ROLE, initParams.emergencyAddress);
}
```

Impact:

Permanent loss of administrative control and inability to upgrade or manage mappings, which can lead to prolonged or permanent DoS of bridging.

Recommendation:

Add explicit non-zero validation for role addresses in `initialize` (similar to Bridge), and consider emitting events for role initialization to ease operational monitoring.

LOW

Mapper.registerMapping does not validate originChainId/targetChainId are non-zero or distinct

Location:

`ether/contracts/main/modules/mapper/Mapper.sol:118-169`

Description:

`Mapper.registerMapping()` validates that the current `block.chainid` matches either `originChainId` (for deposit mappings) or `targetChainId` (for withdrawal mappings), but it does not validate:

- `originChainId != 0` and `targetChainId != 0`
- `originChainId != targetChainId`

Allowing zero or identical chain IDs can create ambiguous/incorrect mappings that may be misinterpreted by off-chain relayers and UIs (e.g., "bridging to the same chain" or "chain 0"), potentially leading to misrouting or operational failures.

Evidence:

Only equality checks are enforced:

```
require(block.chainid == newMapInfo.originChainId, ...);
...
require(block.chainid == newMapInfo.targetChainId, ...);
```

No non-zero/distinctness checks exist.

Impact:

Low: primarily a configuration/input-validation footgun that can result in confusing or incorrect bridge behavior.

Recommendation:

Add explicit bounds/consistency checks on chain IDs (non-zero and distinct) unless same-chain mappings are an intentional feature (in which case document and validate that workflow explicitly).

LOW

Native coin withdrawals can be permanently un-executable if recipient contract rejects ETH (no redirect/recovery path)

Location:

`ether/contracts/main/modules/bridge/Bridge.sol:270-286`

Description:

For native coin mappings (`_mapInfo.isCoin == true`), `receiveTokens()` pays the recipient using a low-level `.call{value: amount}("")` and requires `success == true`. If `toAddress` resolves to a contract that is non-payable or deliberately reverts/consumes all gas in its `receive()`/`fallback()`, the withdrawal transaction will **always revert**.

Because `toAddress` is part of the cross-chain message, the relayer cannot complete the withdrawal without changing the recipient, but the contract provides **no on-chain mechanism** to redirect a failed coin payout to a different address or to allow the intended recipient to pull funds later. The locked coins remain in the bridge contract.

Evidence:

```
(bool success, ) = payable(address(uint160(uint256(receiveTokensParams.toAddress)))).call{
 value: receiveTokensParams.amount
}("");
require(success, "Bridge: Failed to send coins");
```

(Bridge.sol:281-286)

Impact:

- A user can accidentally (or maliciously) specify a destination contract that cannot receive ETH, causing **permanent inability** to finalize that withdrawal on this chain.
- Funds remain locked in the bridge contract unless a privileged actor performs an out-of-band/manual rescue (not implemented as a user-facing recovery path).

Recommendation:

Add a recovery mechanism for failed coin withdrawals (e.g., escrow/pull payments, retry with alternate recipient authorized by original recipient, or an admin-mediated reclaim flow with clear safeguards).

LOW

No reserve/accounting for obligations: multisig liquidity withdrawals can make bridge insolvent and strand users' cross-chain claims

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:251-305`

`ether/contracts/main/modules/bridge/Bridge.sol:340-394`

Description:

The bridge holds **pooled** ETH/ERC20 balances that are used to satisfy `receiveTokens()` unlock transfers. However, it maintains **no on-chain accounting of outstanding obligations** (no per-deposit escrow, no pending-withdraw queue, no reserved liquidity per token).

At the same time, `MULTISIG_ROLE` can withdraw arbitrary amounts of ETH and arbitrary ERC20 balances via `withdrawCoinLiquidity()` / `withdrawTokenLiquidity()`.

Because withdrawals are not reserved, a multisig action (malicious or accidental) can reduce balances below what off-chain relayers consider "owed" to users based on origin-chain locks/burns. When that happens, `receiveTokens()` unlock transfers revert (insufficient ETH after `- gasAccumulated`, or insufficient ERC20 balance / transfer failure), effectively **stranding users' cross-chain claims**.

Evidence:

No obligation tracking exists; `receiveTokens()` only checks current balances:

```
uint256 _contractBalance = address(this).balance - gasAccumulated;
require(_contractBalance >= receiveTokensParams.amount, ...);
// ERC20 path relies on token transfer succeeding
```

Multisig can withdraw liquidity without checking any reserved/owed amount:

```
function withdrawTokenLiquidity(...) external onlyRole(MULTISIG_ROLE) {
 _executeTokenTransfer(..., withdrawTokenLiquidityParams.amount);
}

function withdrawCoinLiquidity(...) external onlyRole(MULTISIG_ROLE) {
 uint256 _contractBalance = address(this).balance - gasAccumulated;
 require(_contractBalance >= withdrawCoinLiquidityParams.amount, ...);
 payable(recipient).call{value: withdrawCoinLiquidityParams.amount}("");
}
```

Impact:

- **High impact**: if users lock/burn assets on the origin chain, but destination-chain liquidity is withdrawn, destination-chain unlocks cannot execute ' users' funds remain locked/burned with no on-chain remedy.
- This is a core asset-management risk in a pooled-liquidity bridge.

Recommendation:

Track and reserve obligations on-chain (per token/coin), and prevent admin liquidity withdrawals from violating reserves. Alternatively, explicitly document that the bridge is fully custodial and withdrawals are discretionary, and add guardrails/timelocks/monitoring to reduce operational errors.

LOW

Role/lifecycle design can permanently lock assets and halt bridging if keys are lost or roles renounced (no on-chain recovery)

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:139-167`

`ether/contracts/main/modules/bridge/Bridge.sol:322-395`

`ether/contracts/main/modules/mapper/Mapper.sol:81-112`

Description:

Critical operations are guarded exclusively by AccessControl roles with no recovery/guardian mechanism:

- Bridge liquidity withdrawals and upgrades are `onlyRole(MULTISIG_ROLE)`.
- Gas fee withdrawal and certain liquidity deposits are `onlyRole(EMERGENCY_ROLE)`.
- Mapper has asymmetric lifecycle controls: `disableMapping()` is `onlyRole(EMERGENCY_ROLE)` while `enableMapping()` is `onlyRole(MULTISIG_ROLE)`.

Because OpenZeppelin AccessControl allows a role-holder to `renounceRole`, and because there is no alternate admin key / timelock / emergency recovery path in these contracts, **key loss or accidental renouncement can permanently lock funds and/or halt bridging**.

Evidence:

- Bridge roles assigned during initialization, with `DEFAULT_ADMIN_ROLE` granted only to `multisigAddress`:

(Bridge.sol:139-167)

- Only multisig can withdraw token/coin liquidity:

(Bridge.sol:340-395)

- Only emergency can withdraw `gasAccumulated` and deposit liquidity via `depositTokens/depositCoins`:

(Bridge.sol:322-445)

- Mapper: emergency can disable mappings, but only multisig can re-enable:

(Mapper.sol:94-112)

Impact:

- **Stuck value**: if MULTISIG_ROLE is lost, token/coin liquidity cannot be recovered; if EMERGENCY_ROLE is lost, gas fee funds and emergency liquidity operations may be stuck.
- **Bridge liveness failure**: mappings can be disabled (intentionally or accidentally) and become impossible to re-enable if multisig is unavailable.

Recommendation:

Introduce explicit recovery mechanisms appropriate to the protocol's trust model, e.g.:

- multiple admins / role guardian that can re-grant roles after delay
- timelocked role changes
- emergency "break-glass" recovery or upgrade path
- on-chain procedures preventing complete role lockout

INFORMATIONAL

Bridge ECDSA authorization hash lacks domain separation and does not bind mapId/originTokenAddress

Location:

`ether/contracts/main/modules/bridge/Bridge.sol:622-659`

Description:

`Bridge._validateECDSA()` constructs the signed message hash with `abi.encodePacked` over a subset of fields. The hash **does not include**:

- `address(this)` (verifying contract)
- `block.chainid` (explicit domain separation)
- `bridgeParams.mapId`
- `mapInfo.originTokenAddress`

This enables: 1) **Cross-contract replay on the same chain**: the same relayer signature can be reused on any other Bridge deployment on the same chain that trusts the same relayer key, because `usedHashes` is per-contract and the hash is identical. 2) **Map substitution**: a signature intended for one mapping can be reused with a different `mapId` as long as the included fields (`targetTokenAddress`, chain IDs, amounts, etc.) match. Since `originTokenAddress` is not part of the signed payload, authorization can be applied to a different origin asset.

This matches Phase-1 suspect `broken-authentication-and-replay` (source: database).

Vulnerable Code:

```
hash = keccak256(
 abi.encodePacked(
 _msgSender(),
 bridgeTokensParams.bridgeParams.toAddress,
 mapInfo.targetTokenAddress,
 gasAmount,
 bridgeTokensParams.bridgeParams.amount,
 mapInfo.originChainId,
 mapInfo.targetChainId,
 bridgeTokensParams.ECDSAParams.deadline,
 bridgeTokensParams.ECDSAParams.salt
 )
);
```

Concrete Exploit Sketch (map substitution):

Assume two deposit mappings exist on the origin chain:

- `mapId=1`: originToken=TokenA, targetToken=TokenT, originChainId=X, targetChainId=Y
- `mapId=2`: originToken=TokenB, targetToken=TokenT, originChainId=X, targetChainId=Y

A relayer signs a message authorizing a deposit for `mapId=1`. Because `mapId` and `originTokenAddress` are not signed, the same signature is valid when the user calls `bridgeTokens()` with `mapId=2` (hash remains identical). The bridge will lock/burn TokenB, yet off-chain processing may mint/unlock TokenT on the target chain.

Impact:

Depending on relayer/off-chain processing assumptions, this can lead to:

- Bypassing relayer authorization intent (signature reuse across deployments)
- Bridging the wrong origin asset under a valid signature (potentially minting/unlocking unbacked target assets if the mapping set allows multiple origins to the same target)
- Operational failures during upgrades/migrations (old signatures unexpectedly valid on new deployments)

Recommendation:

- Use EIP-712 domain separation including `address(this)` and `block.chainid`.
- Include `bridgeParams.mapId` and/or `mapInfo.originTokenAddress` (and any other critical mapping attributes such as `isCoin/depositType`) in the signed payload.
- Consider binding to a monotonically increasing nonce per user instead of relying on `salt` + `usedHashes` per-contract.

INFO

Comments/documentation mismatch about who can deposit and what removeMapping validates

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:396-444`

`ether/contracts/main/modules/mapper/Mapper.sol:171-191`

Description:

Several comments/documentation blocks do not match the implemented access control / validation, which is a correctness and maintenance hazard (operators may rely on the comments).

Evidence:

Bridge.depositTokens / depositCoins comments say "Allows users…", but only EMERGENCY_ROLE can call:

```
/**
 * @notice Allows users to deposit tokens into the contract.
 * ...
 * Only the address with EMERGENCY role can call this function.
 */
function depositTokens(...) external ... onlyRole(EMERGENCY_ROLE) ...
```

(Bridge.sol:396-435)

Similarly:

```
function depositCoins() external payable onlyRole(EMERGENCY_ROLE) ...
```

(Bridge.sol:437-444)

Mapper.removeMapping comment claims it validates chain ownership, but implementation does not:

The comment says it "ensures that the provided mapId is valid and belongs to the current chain", but `removeMapping` only checks `mapCounter >= mapId` and does not verify `block.chainid` matches the mapping's origin/target chain. (Mapper.sol:171-191)

Impact:

Operational misunderstanding can lead to incorrect procedures (e.g., expecting public deposits), misconfiguration, and increased risk during incident response.

Recommendation:

Update comments to reflect actual behavior, or adjust code to match documented intent.

INFO

ECDSAChecks.recoverSigner hardcodes Ethereum personal_sign prefix and does not normalize v, which can reject otherwise valid signatures

Location:

`ether/contracts/main/libraries/ECDSAChecks.sol:39-43`

Description:

`ECDSAChecks.recoverSigner()` always wraps the provided digest with `toEthSignedMessageHash()` (the EIP-191 `"\x19Ethereum Signed Message:\n32"` prefix) and then calls OZ `recover(v,r,s)`.

This is semantically specific to Ethereum-style `personal_sign` / `signMessage(arrayify(hash))` flows. If signers produce signatures over:

- the *raw* digest without the prefix (`eth_sign` style in some stacks),
- EIP-712 typed data,
- or use recovery IDs `v` as `0/1` (common output of some libraries),

then verification will fail even though the signatures are otherwise valid secp256k1 signatures.

Evidence:

```
bytes32 _messageHash = _ECDSAParams.hash.toEthSignedMessageHash();

signer = _messageHash.recover({ v: _ECDSAParams.v, r: _ECDSAParams.r, s: _ECDSAParams.s });
```

(ECDSAChecks.sol:39-43)

Impact:

Operational/integration risk: relayer signatures may be rejected unexpectedly depending on the signing method and `v` convention used off-chain.

Recommendation:
Document the required signing procedure precisely (e.g., `signMessage(arrayify(hash))` with v=27/28), or add explicit normalization/support for other signature formats if needed.

INFO

ERC165 supportsInterface checks in Bridge can be spoofed and may not provide the intended safety against wrong/malicious Mapper addresses

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:153-167`

`ether/contracts/main/modules/bridge/Bridge.sol:452-466`

Description:

`Bridge.initialize()` and `Bridge.changeMapperAddress()` attempt to validate the Mapper contract using `ERC165Checker.supportsInterface(..., type(IMapper).interfaceId)`.

ERC165 signaling is not a correctness/security guarantee: any contract can implement `supportsInterface()` to return `true` without actually behaving as a correct Mapper. If the intended purpose of this check is to prevent misconfiguration (setting a non-Mapper address) or to provide a safety boundary, it does not fully achieve that intent.

Evidence:

```
require(
 ERC165Checker.supportsInterface({
 account: initParams.mapperAddress,
 interfaceId: type(IMapper).interfaceId
 }),
 "Bridge: New address does not support IMapper"
);
```

(Bridge.sol:153-159)

and similarly in `changeMapperAddress()` (Bridge.sol:455-458).

Impact:

Operational/correctness risk: the bridge may accept a Mapper address that only *claims* ERC165 support. Subsequent calls (e.g., `Mapper.mapInfo(mapId)`) can return arbitrary data, leading to incorrect asset handling.

Recommendation:

Treat ERC165 checks as best-effort UX only, not as a security boundary. Consider stronger validation (e.g., immutable mapper, vetted upgrade path, or additional sanity checks on returned mapping data) if the intention is to prevent wrong-contract configuration.

INFO

Mapper MULTISIG_ROLE description contradicts implementation (cannot disable mappings without EMERGENCY_ROLE)

Locations:

`ether/contracts/main/modules/mapper/Mapper.sol:22-27`

`ether/contracts/main/modules/mapper/Mapper.sol:94-112`

Description:

`Mapper` documents `MULTISIG_ROLE` as “responsible for enabling and disabling mappings” (contract-level comment), but `disableMapping()` is restricted to `onlyRole(EMERGENCY_ROLE)`.

While `DEFAULT_ADMIN_ROLE` (multisig) could grant itself `EMERGENCY_ROLE`, the stated responsibility and the enforced access control are inconsistent.

Evidence:

```
// MULTISIG_ROLE: "responsible for enabling and disabling mappings." (comment)
bytes32 public constant MULTISIG_ROLE = keccak256("MULTISIG_ROLE");

function enableMapping(uint256 mapId) external onlyRole(MULTISIG_ROLE) { ... }
function disableMapping(uint256 mapId) external onlyRole(EMERGENCY_ROLE) { ... }
```

Impact:

Informational: may lead to incorrect operational assumptions about which key needs to be used for emergency shutdown vs normal mapping toggles.

Recommendation:

Align comments/docs with actual access control, or adjust role gating to match intended responsibilities.

INFO

Open receive() emits CoinsDeposited with no bridge context, which can be misused/abused by off-chain indexers as a “deposit” signal

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:125-133`

`ether/contracts/main/modules/bridge/interfaces/IBridge.sol:209-214`

Description:

`Bridge.receive()` is an unpermissioned ETH receive hook that emits `CoinsDeposited(account, amount)` for **any** plain ETH transfer.

This event contains no `mapId`, no `toAddress`, no chain IDs, and no signature context. If off-chain relayers/indexers (now or in the future) mistakenly treat `CoinsDeposited` as a bridge-deposit trigger (instead of using the `Deposit` event from `bridgeTokens()`), an attacker could mint/release assets on the target chain by simply sending ETH to the contract (bypassing the relayer-signed `bridgeTokens()` flow and its replay protection).

Even if the current relayer implementation ignores this event, it is an economic-foot-gun integration surface because `CoinsDeposited` is emitted both for the privileged `depositCoins()` and for arbitrary user transfers, making it easy to misinterpret.

Evidence:

```
receive() external payable {
 emit CoinsDeposited({ account: _msgSender(), amount: msg.value });
}
```

Impact (Economic):

Potential off-chain accounting/minting bypass if `CoinsDeposited` is used as an input to cross-chain issuance/release logic. This is a common class of bridge exploit: ambiguous/overloaded events.

Recommendation:
Ensure off-chain systems only use the `Deposit` event from `bridgeTokens()` for bridging. Consider renaming/removing `CoinsDeposited` from `receive()`, or emitting a distinct event for “liquidity/top-up” vs “bridge deposit” to reduce integration risk.

INFO

Privileged actions/upgrades are immediate with no timelock (governance key compromise = instant catastrophic changes)

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:452-519`

`ether/contracts/main/modules/mapper/Mapper.sol:94-211`

Description:

Critical privileged operations execute immediately with only role checks and no on-chain delay/timelock, including:

- Bridge: `changeMapperAddress`, `setDailyLimit`, `_authorizeUpgrade`, liquidity withdrawals
- Mapper: `registerMapping`, `enableMapping`, `_authorizeUpgrade`, and EMERGENCY disables/removals

This matches Phase-1 suspect `vp_governance_Mapper_6bd2bb96`.

Impact:

If MULTISIG_ROLE/EMERGENCY_ROLE is compromised or acts maliciously, the attacker can instantly:

- Upgrade implementations, change mappings, and withdraw liquidity
- Disrupt bridging or mint/unlock to arbitrary recipients (via role reassignment on upgrade)

Recommendation:

Consider adding a timelock or staged changes (propose/accept) for upgrades and critical parameter changes, and separating duties/thresholds for EMERGENCY vs MULTISIG.

INFO

Signature hash packing uses uint64 deadline; off-chain encoders using uint256 will produce unverifiable signatures

Locations:

`ether/contracts/main/modules/bridge/Bridge.sol:622-644`

`ether/contracts/main/modules/bridge/interfaces/IBridge.sol:34-40`

Description:

`Bridge._validateECDSA()` builds the signed preimage with `abi.encodePacked(..., bridgeTokensParams.ECDSAParams.deadline, ...)`.

In `IBridge.ECDSAParams`, `deadline` is declared as `uint64`.

When using `abi.encodePacked`, `uint64` is encoded as **8 bytes**, not 32. Off-chain implementations that naively ABI-pack deadline as a 32-byte `uint256` (a common default) will compute a different hash and produce signatures that always fail on-chain.

Evidence:

```
// IBridge
struct ECDSAParams {
 ...
 uint64 deadline;
 uint8 v;
}

// Bridge
hash = keccak256(abi.encodePacked(
 ...,
 bridgeTokensParams.ECDSAParams.deadline, // 8-byte packed field
 bridgeTokensParams.ECDSAParams.salt
));
```

Impact:

Operational fragility: signatures may be rejected depending on how relayers encode `deadline` off-chain.

Recommendation:

Ensure off-chain signers explicitly pack `deadline` as uint64 (8 bytes) to match on-chain encoding, or change the on-chain encoding to a less error-prone scheme (e.g., `abi.encode` with fixed 32-byte words) and version the signing domain.

Advisory to Clients

Cecuro highly recommends scheduling a follow-up security assessment within six months of this report or immediately after any significant modifications to the codebase, whichever occurs first. This practice is essential for preserving the project's security posture and identifying potential vulnerabilities that may arise from code changes or updates.

Disclaimer

The smart contracts submitted for analysis have been reviewed using Cecuro.ai's AI security analysis system, applying industry-standard vulnerability detection patterns and best practices at the time of this report. The findings disclosed in this report reflect the AI system's assessment of the submitted source code.

This report contains no statements or warranties on the identification of all vulnerabilities or the overall security of the code. Any security review methodology may not detect all potential issues, particularly novel attack vectors, complex business logic flaws, or economic exploits. The report covers the code submitted at the specified version and may not be relevant after any modifications.

Do not consider this report as a final and sufficient assessment regarding the security, utility, or bug-free status of the code. We recommend complementing this analysis with additional independent audits and a public bug bounty program to strengthen the security posture of your smart contracts.

Smart contracts are deployed and executed on blockchain platforms. The platform, its programming language, compiler, and other related software may contain vulnerabilities beyond the scope of code-level analysis. Cecuro.ai cannot guarantee the explicit security of the analyzed smart contracts.

This report does not constitute financial, legal, or investment advice. It is not a recommendation to buy, sell, or invest in any token or project, nor an endorsement of any kind. Users should conduct their own independent due diligence.

cecuro

Agentic Smart Contract Auditing