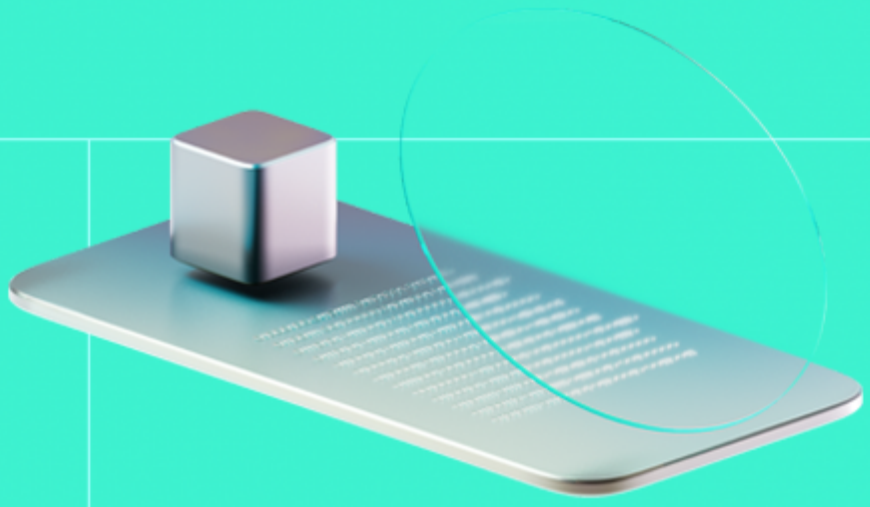
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Makachain

Date: 03/06/2026

We express our gratitude to the Makachain team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Makachain Contracts provides a cross-chain bridge infrastructure built on the Hyperlane messaging protocol, targeting the Makachain network. A modified Hyperlane synthetic ERC-20 token is extended with a flat fee mechanism — denominated in the transferred token's own units — that is deducted on every on-chain transfer, with fee proceeds coordinated through a centralized oracle.

Document

Name	Smart Contract Code Review and Security Analysis Report for Makachain
Audited By	Khrystyna Tkachuk
Approved By	Kornel Światłowski
Website	makachain.io
Changelog	26/05/2026 - Preliminary Report
	03/06/2026 - Final Report
Platform	Makachain
Language	Solidity
Tags	ERC20; Upgradable; Bridge; Centralization
Methodology	https://docs.hacken.io/methodologies/smart-contracts

Review Scope

Repository	https://github.com/MakaChain_contracts
Commit	e6437f5
Final Commit	7442225

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

14	9	5	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	1
Medium	3
Low	3

Vulnerability	Severity	Status
F-2026-17370 - Pool Insolvency Causes Transfer Denial-of-Service	High	Fixed
F-2026-17372 - Authorization-Fee Conflation Bricks All Transfers When MAKA Fee Is Set to Zero	Medium	Fixed
F-2026-17373 - Wrong Token Used in chargeMaka Fallback Path Results in Incorrect Asset Distribution	Medium	Fixed
F-2026-17390 - Flat Fee Without Amount Validation in _transfer Causes Silent Zero-Value Transfers and DoS on Small Amounts	Medium	Fixed
F-2026-17377 - Missing Zero-Address Validation for _owner Parameter in initialize May Lead to Loss of Administrative Control	Low	Accepted
F-2026-17392 - _isEOA Bypass via extcodesize During Construction Enables Gas Subsidy Extraction	Low	Accepted
F-2026-17395 - Lack of Admin Withdraw Function for Excess MAKA Tokens	Low	Fixed
F-2026-17369 - Floating Pragma	Info	Accepted
F-2026-17374 - Lack of _disableInitializers Call in Constructor	Info	Accepted
F-2026-17375 - Unused Event and Custom Error Declaration	Info	Fixed
F-2026-17376 - Lack of Event Emitting for Important State Changes in constructor	Info	Fixed
F-2026-17378 - Redundant Default Value Initialization	Info	Fixed
F-2026-17393 - Unchecked Return Value in chargeMaka Fallback Path Causes Silent Transfer Failure	Info	Fixed
F-2026-17394 - Lack of Two Step Ownership Transfer Mechanism	Info	Accepted

Documentation quality

- Functional requirements are partially missed.
- Technical description are partially provided.
- NatSpec comments are sufficient.
- Expected behavior for edge cases is not documented.

Code quality

- The code leverages OpenZeppelin contracts and follows established patterns.
- The development environment is not configured.

Test coverage

Code coverage of the project is **0%**.

- No test suite was provided by the client.

Table of Contents

System Overview **6**

Files In Scope 6

Privileged Roles

Potential Risks **8**

Findings **10**

Vulnerability Details 10

Disclaimers 46

Appendix 1. Definitions **47**

Severities 47

Potential Risks 47

Appendix 2. Scope **48**

Appendix 3. Additional Valuables **49**

System Overview

The bridge subsystem is composed of two contracts that collectively implement a fee-bearing cross-chain synthetic token. **HypERC20** is a modified Hyperlane **TokenRouter** / **ERC20Upgradeable** synthetic token whose `_transfer` override intercepts every token movement. When an oracle address is configured, the override first instructs **TrustedOracle** to perform a native-coin gas subsidy via `gasBack`, then queries it for the applicable fee. If a non-zero fee is returned and passes the `require(feeAmount < amount)` validation, the transfer is split into a fee portion (sent to the oracle) and a net portion (sent to the transfer recipient). When the oracle is not set, transfers fall through to the standard ERC-20 behaviour unchanged. Cross-chain minting and burning are inherited unmodified from the upstream Hyperlane **TokenRouter** stack, which routes messages through a Hyperlane **Mailbox** and **InterchainSecurityModule**.

TrustedOracle serves as the fee authority and the single administrative hub. Per-token flat fee amounts (in the transferred token's denomination), per-token whitelist administrators, and admin-controlled price feeds stored by `bytes32` symbol are maintained as configuration state. Fee calculation via `calculateFee` returns zero when either party is the zero address, the configured recipient, or a whitelisted address for the given token, and otherwise returns the flat fee for that token. A `gasBack` function automatically tops up the native-coin balance of participating EOAs to a 0.2 ether threshold, sourcing funds through a chain-specific mint precompile when the oracle's own native balance falls below 1,000 ether. Access to `gasBack` is gated by an explicit `registeredContracts` allowlist managed by the admin via `setRegisteredContracts`, decoupling authorization from fee configuration. Core configuration — fees, prices, the MAKA token address, token administrators, and the fee recipient — is managed exclusively by a single admin address, while per-token whitelist management and fee withdrawal are delegated to individually assigned token administrators.

Files in Scope

- **HypERC20Flat.sol** — Contains the custom **HypERC20** contract (the sole modified contract in a flattened upstream Hyperlane token stack). Inheritance from **ERC20Upgradeable** and **TokenRouter** is used, `_transfer` is overridden to intercept transfers and route fees through an oracle with an explicit `require(feeAmount < amount)` validation, and `setOracle` is provided for the contract owner to enable or disable the fee mechanism.
- **TrustedOracle.sol** — The centralized fee and configuration authority. Per-token flat fees are computed via `calculateFee` (respecting whitelist exemptions, zero-address guards, and recipient-address exemptions). Automatic native-coin gas subsidies are provided via `gasBack` (gated by the `registeredContracts` allowlist), and admin-only setters are exposed for fees, prices, whitelists, token admins, registered contracts, and the recipient.

Privileged roles

HypERC20 (HypERC20Flat.sol):

- **owner** (inherited from **OwnableUpgradeable** via **MailboxClient → Router → GasRouter → TokenRouter**): Controls all administrative functions of the Hyperlane token router.

- Can call `setOracle` to set or disable the oracle address used for transfer-fee calculation and gas-back logic.
- Can call `setHook` to replace the post-dispatch hook contract.
- Can call `setInterchainSecurityModule` to replace the interchain security module contract.
- Can call `enrollRemoteRouter` to register a trusted remote router for a given destination domain.
- Can call `enrollRemoteRouters` to batch-register trusted remote routers for multiple destination domains.
- Can call `unenrollRemoteRouter` to remove a trusted remote router for a given destination domain.
- Can call `unenrollRemoteRouters` to batch-remove trusted remote routers for multiple destination domains.
- Can call `setDestinationGas` to configure the gas limit for cross-chain dispatches to a specific domain (single or batch).
- Can call `setFeeRecipient` to set the address that receives Hyperlane router-level transfer fees.
- Can call `setFeeHook` to set the fee hook contract used for external fee calculation.
- Can call `transferOwnership` to transfer the owner role to a new address.
- Can call `renounceOwnership` to irrevocably renounce the owner role, disabling all owner-gated functions.

TrustedOracle.sol

- **admin**: Full administrative control over fee configuration, pool assignments, pricing, and role management.
  - Can call `setFees` to configure per-token transfer fees (batch).
  - Can call `setRegisteredContracts` to explicitly register or deregister contract addresses authorized to call `gasBack` (batch).
  - Can call `setRecipient` to set the recipient address used in fee-exemption checks.
  - Can call `setTokenAdmins` to assign token-specific admin addresses (batch).
  - Can call `setPrices` to set prices for token symbols (batch).
  - Can call `transferAdmin` to transfer the admin role to a new address.
- **tokenAdmin** (per-token, assigned by **admin** via `setTokenAdmins`): Per-token administrative role for whitelist management and fee withdrawal.
  - Can call `setWhitelist` to add or remove addresses from the fee-exemption whitelist for the token they administer (batch).
  - Can call `withdrawFees` to withdraw accumulated fee balances of the administered token held by the oracle contract.
- **registeredContract** (explicit; managed via `setRegisteredContracts` by the admin): Authorized to invoke gas-back operations on behalf of integrated tokens.
  - Can call `gasBack` to auto-refill native coin balances of EOA addresses involved in a transfer, minting native coins via the **MintPrecompile** if the oracle's own balance is low.

Potential Risks

- **Dependency on Hyperlane Framework Inheritance Chain**: **HypERC20** inherits through a deep chain — **TokenRouter → GasRouter → Router → MailboxClient → OwnableUpgradeable** — all embedded in the flattened file as Hyperlane framework code. Critical behaviours such as cross-chain message dispatch via `_Router_dispatch`, inbound token minting via `_handle`, remote router enrollment via `enrollRemoteRouter`, and the `onlyOwner` access control modifier are defined entirely in these parent contracts. Any latent defect in this inherited logic would propagate into **HypERC20** without being addressed by the current scope.
- **System Reliance on the Hyperlane Mailbox**: All cross-chain functionality in **HypERC20** depends on an immutable **IMailbox** address set in the **MailboxClient** constructor and used by `_Router_dispatch` for outbound message dispatch and the `onlyMailbox` modifier for inbound message handling. The **Mailbox** contract is external to the repository and cannot be upgraded or replaced without redeploying the entire **HypERC20** proxy. Unavailability, compromise, or misconfiguration of the **Mailbox** would halt all bridge operations.
- **MakaChain Native-Coin Mint Precompile Dependency**: The `gasBack` function in **TrustedOracle** invokes `mintNativeCoin` on a hardcoded precompile at address `0x0200000000000000000000000000000000000001` to refill the contract's native-coin balance when it falls below 1,000 ether. This is a MakaChain-specific infrastructure component whose behaviour, availability, and access-control semantics are defined at the chain level and are not auditable from Solidity source. If the precompile interface changes, is deprecated during a chain upgrade, or enforces caller restrictions not currently present, the gas-subsidy mechanism would revert, blocking all transfers on **HypERC20** for which the oracle is configured and the contract's native balance has been depleted.
- **Fixed Total Supply Post-Deployment**: The token's total supply is determined at deployment and cannot be verified beforehand, potentially limiting the project's adaptability and economic model flexibility.
- **Single Points of Failure and Control**: The project is fully or partially centralized, introducing single points of failure and control. This centralization can lead to vulnerabilities in decision-making and operational processes, making the system more susceptible to targeted attacks or manipulation.
- **Absence of Timelock Mechanisms on All Critical Operations**: Every administrative function across **HypERC20** and **TrustedOracle** executes immediately upon transaction confirmation. Operations such as `setOracle`, `setFees`, `setRegisteredContracts`, `setPrices`, `setInterchainSecurityModule`, `enrollRemoteRouter`, and `transferAdmin` take effect in the same block, providing no window for users or monitoring systems to detect and react to potentially harmful parameter changes before they impact live transfers.
- **Arbitrary Oracle Address Setting on HypERC20**: The **HypERC20** owner can replace the `oracle` address at any time via `setOracle` with no validation, timelock, or multi-signature requirement. The oracle intercepts every ERC-20 `_transfer` call to execute `gasBack` and `calculateFee`, meaning a malicious or incorrect oracle address could extract fees, mint arbitrary native coins, or revert all token transfers chain-wide.
- **Unrestricted Fee and Price Modification by TrustedOracle Admin**: The **TrustedOracle** `admin` can call `setFees` to set arbitrary `feesForToken` values per token, `setPrices` to set arbitrary token symbol prices, and `setRecipient` to redirect fee proceeds — all without bounds checking, rate

limits, or timelocks. A compromised or malicious admin key could set prohibitively high fees to effectively freeze transfers or redirect fee revenue to an attacker-controlled address.

- **Centralized Arbitrary Token Admin Setting:** The **TrustedOracle** contract `admin` can assign arbitrary addresses as token admins for any token via `setTokenAdmin` without restrictions on the number of admins or token coverage, granting each appointed admin the ability to withdraw any amount of that token from the oracle contract. A compromised or malicious `admin` can appoint colluding addresses that collectively drain all token balances held by the oracle, with no on-chain safeguards such as timelocks, multisig requirements, or withdrawal caps to limit the damage.
- **Centralized Fee Oracle Without Fallback or External Validation**: **TrustedOracle** serves as the sole fee-calculation authority for all **HypERC20** transfers, with fee amounts set exclusively by the `admin` via `setFees` and no secondary oracle, on-chain price feed, or sanity-check bounds to validate or override these admin-supplied values. If the admin key becomes unavailable, there is no fallback mechanism, and fee parameters will remain at whatever values were last configured.
- **Uncapped Fund Drainage:** The `withdrawFees` function lacks fee accounting, allowing the token admin to withdraw the entire oracle contract balance.
- **Flexibility and Risk in Contract Upgrades**: The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.
- **Explicit Contract Registration Requirement**: The `onlyRegisteredContract` modifier in **TrustedOracle** authorizes callers via an explicit `registeredContracts` allowlist managed by the admin through `setRegisteredContracts`. `gasBack` — the sole function gated by this modifier — is called unconditionally by `HypERC20._transfer` when an oracle is configured, meaning any **HypERC20** token contract must be registered before transfers will succeed. If a token contract is not registered (e.g., after an upgrade deploys a new oracle version with an empty mapping, or if the admin deregisters a token), all transfers on that **HypERC20** instance will revert with `Unauthorized()` until registration is restored. Conversely, registering any arbitrary address grants it permission to trigger native-coin minting through `gasBack`, bounded by the per-EOA 0.2 ether subsidy cap.

Findings

Vulnerability Details

F-2026-17370 - Pool Insolvency Causes Transfer Denial-of-Service - High

Description:

HypERC20 is a fee-bearing cross-chain synthetic ERC20 token that extends the standard Hyperlane token transfer flow with oracle-based token fee charging and MAKA fee distribution. During regular transfers, the contract deducts a configured token fee and then triggers `TrustedOracle.chargeMaka()` to distribute MAKA from the associated **Pool** to dedicated `recipient` address configured by admin.

The token transfer flow depends on successful MAKA distribution from the configured **Pool** contract. During each fee-bearing transfer, `HypERC20._transfer()` calls `TrustedOracle.chargeMaka()`, which then calls `Pool.chargeMaka()` to transfer a fixed amount of MAKA from the **Pool** to the recipient.

```
// HypERC20Flat.sol
function _transfer(address from, address to, uint256 amount) internal virtual
override {
    // Skip fee if oracle not set
    if (oracle != address(0)) {

        //gasBack
        ITrustedOracle(oracle).gasBack(from, to);

        // Get fee from oracle (in wei)
        uint256 feeAmount = ITrustedOracle(oracle).calculateFee(address(this)
, from, to, amount);

        if (feeAmount > 0) {
            // Transfer fee amount to oracle
            super._transfer(from, oracle, feeAmount);

            // Transfer remaining amount to recipient
            uint256 amountAfterFee = amount - feeAmount;
            super._transfer(from, to, amountAfterFee);

            ITrustedOracle(oracle).chargeMaka(address(this), from, to, amount
); <-- revert on lack of MAKA liquidity
```

```
            return;
        }
    }

// TrustedOracle.sol
function chargeMaka(address token, address from, address to, uint256 amount)
external onlyRegisteredContract {
    // Get pool for this token
    address pool = pools[token];

    // Get MAKA amount to charge
    uint256 makaAmount = feesForMaka[token];

    if (pool != address(0) && recipient != address(0) && makaAmount > 0) {
        IPool(pool).chargeMaka(makaToken, recipient, makaAmount);
        return;
    }
    if (recipient != address(0) && makaAmount > 0) {
        IERC20(token).transfer(recipient, makaAmount);
        return;
    }
}

// Pool.sol
function chargeMaka(address makaToken, address recipient, uint256 makaAmount)
external onlyOracle {
    if (makaAmount == 0) revert InvalidAmount();

    // Check pool has enough MAKA liquidity
    uint256 poolBalance = IERC20(makaToken).balanceOf(address(this));
    if (poolBalance < makaAmount) revert InsufficientLiquidity();

    // Transfer MAKA from pool to recipient
    require(IERC20(makaToken).transfer(recipient, makaAmount), "Transfer out
failed");

    emit MakaCharged(msg.sender, makaToken, recipient, makaAmount);
}
```

However, the Pool's MAKA balance is continuously depleted by transfers and there is no enforced on-chain refill mechanism, reserve threshold, or graceful fallback when the **Pool** becomes insolvent. Once the **Pool** balance is lower than the configured `feesForMaka[token]` amount, `Pool.chargeMaka()` reverts with `InsufficientLiquidity()`. This revert propagates unhandled through **TrustedOracle** (no try/catch) into **HypERC20**'s `_transfer`, blocking all fee-bearing transfers for every token mapped to the depleted pool.

The issue is amplified when multiple tokens are mapped to the same **Pool**. In that case, transfers of one token can drain the shared **Pool** and block transfers of all other tokens using the same **Pool**. This allows a low-value or attacker-controlled token to exhaust the MAKA liquidity used by higher-value tokens, causing cross-token denial of service.

This lead to the following consequences:

- A depleted MAKA **Pool** causes a complete denial of service for all fee-bearing token transfers that depend on that **Pool**.
- Affected users cannot transfer tokens because the transfer flow reverts during MAKA fee distribution. Only addresses that are exempt from fees, where `feeAmount == 0` and `chargeMaka()` is skipped, remain functional.
- Cross-chain minting and burning flows are not affected as they bypass `_transfer()`. However, regular ERC20 transfers remain blocked until an external actor manually refills the **Pool** with sufficient MAKA liquidity.
- The only recovery path is a manual off-chain ERC20 transfer to the Pool address or setting the oracle address to zero via `setOracle()`. However, disabling the oracle also disables fee accumulation entirely, which is not a graceful degradation mechanism.

An attacker can accelerate the condition by repeatedly executing small fee-bearing transfers. Each transfer pays only the token fee and gas cost while draining a fixed amount of MAKA from the **Pool**.

Assets:

- src/bridge/HypERC20Flat.sol [https://github.com/MakaChain/contracts]
- src/bridge/TrustedOracle.sol [https://github.com/MakaChain/contracts]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 4/5

Exploitability: Independent

Complexity: Simple

Severity: High

Recommendations

Remediation: Decouple MAKA fee distribution from the core token transfer flow. A failure to distribute MAKA should not cause the user's token transfer to revert.

Additionally, consider implementing a minimum **Pool** balance threshold in the oracle or **Pool** logic. If the **Pool** balance is below the required MAKA amount, the system should skip MAKA distribution and emit an event for off-chain monitoring instead of reverting the token transfer.

This ensures that a MAKA fee distribution failure does not compromise the availability of the token transfer mechanism, while still enforcing the token fee.

Resolution: **Fixed in `76e16c8`.**

The MAKA distribution path that triggered the revert has been removed entirely. `TrustedOracle.chargeMaka()` and the `IPool` interface were deleted from **TrustedOracle**, and the `ITrustedOracle(oracle).chargeMaka()` call was removed from `HypERC20._transfer()`, alongside deletion of `bridge/Pool.sol`. With no pool call inside the transfer flow, the `InsufficientLiquidity()` revert can no longer propagate into `_transfer()` and block fee-bearing transfers.

Evidences

Pool Insolvency Causes Transfer Denial-of-Service

Reproduce: **PoC steps:**

1. Admin deploys `HypERC20`, `TrustedOracle`, `Pool`, and `MockERC20` (MAKA token)
2. Admin transfers 10,000 tokens to `user1` before enabling the oracle (no fees charged)
3. Admin configures the oracle with a 1 MAKA fee per transfer and seeds the pool with exactly 3 MAKA
4. `user1` performs 3 transfers of 100 tokens to `user2`, each draining 1 MAKA from the pool
5. Pool MAKA balance reaches 0
6. `user1` attempts a 4th transfer and it reverts with `InsufficientLiquidity()`
7. `user1` still holds sufficient token balance but cannot transfer

The PoC demonstrates that every non-whitelisted token transfer drains a fixed MAKA amount from the pool, and the pool has no replenishment mechanism. Once the seeded MAKA is exhausted, all token transfers revert permanently, causing a protocol-wide DoS despite users holding sufficient token balances.

PoC code:

```
contract PoC_PoolInsolvencyDoS is Test {
    TrustedOracle oracle;
    Pool pool;
    HypERC20 token;
```

```
    MockERC20 maka;

    address admin = makeAddr("admin");
    address user1 = makeAddr("user1");
    address user2 = makeAddr("user2");
    address feeRecipient = makeAddr("feeRecipient");

    uint256 constant MAKA_FEE = 1 ether;
    uint256 constant TOKEN_FEE = 0.01 ether;
    uint256 constant POOL_SEED = 3 ether;

    function setUp() public {
        vm.startPrank(admin);

        MockMailbox mailbox = new MockMailbox();
        maka = new MockERC20();
        oracle = new TrustedOracle();
        pool = new Pool(address(oracle));
        token = new HypERC20(18, 1, 1, address(mailbox));
        token.initialize(1_000_000 ether, "TestToken", "TT", address(0), addr
ess(0), admin);

        // Fund user1 BEFORE enabling oracle so this transfer doesn't drain t
he pool
        token.transfer(user1, 10_000 ether);

        vm.deal(address(oracle), 2000 ether);

        token.setOracle(address(oracle));
        oracle.setMakaToken(address(maka));
        oracle.setRecipient(feeRecipient);

        address[] memory tokens = new address[](1);
        tokens[0] = address(token);
        uint256[] memory tokenFees = new uint256[](1);
        tokenFees[0] = TOKEN_FEE;
        uint256[] memory makaFees = new uint256[](1);
        makaFees[0] = MAKA_FEE;
        oracle.setFees(tokens, tokenFees, makaFees);

        address[] memory poolAddrs = new address[](1);
        poolAddrs[0] = address(pool);
        oracle.setPools(tokens, poolAddrs);

        maka.mint(address(pool), POOL_SEED);

        vm.stopPrank();

        vm.deal(user1, 1 ether);
```

```
        vm.deal(user2, 1 ether);
    }

    function test_poolDepletionBlocksTransfers() public {
        uint256 transferAmount = 100 ether;

        for (uint256 i = 0; i < 3; i++) {
            vm.prank(user1);
            token.transfer(user2, transferAmount);
        }
        assertEq(maka.balanceOf(address(pool)), 0, "Pool fully drained");

        vm.prank(user1);
        vm.expectRevert(abi.encodeWithSignature("InsufficientLiquidity()"));
        token.transfer(user2, transferAmount);

        assertGt(token.balanceOf(user1), transferAmount, "user1 has tokens bu
t transfers are blocked");
    }
}
```

Results:

```
Ran 1 test for test/PoC_PoolInsolvencyDoS.t.sol:PoC_PoolInsolvencyDoS
[PASS] test_poolDepletionBlocksTransfers() (gas: 220848)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.38ms (882.63µs
CPU time)

Ran 1 test suite in 111.63ms (7.38ms CPU time): 1 tests passed, 0 failed, 0 s
kipped (1 total tests)
```

Files: PoC_PoolInsolvencyDoS.t.sol

F-2026-17372 - Authorization-Fee Conflation Bricks All Transfers When MAKA Fee Is Set to Zero - Medium

Description:

HypERC20 is a fee-bearing cross-chain synthetic token that integrates with **TrustedOracle** during token transfers. On each transfer, `HypERC20._transfer()` calls the oracle's `TrustedOracle.gasBack()` function when the oracle address is configured, allowing the oracle to refill native coin balances for the `from` and `to` addresses.

```
function gasBack(address from, address to) external onlyRegisteredContract {
    // Check contract balance and top up to 5000 if below 1000
    if (address(this).balance < 1000 ether) {
        IMintPrecompile(0x0200000000000000000000000000000000000001).mintNativ
eCoin(address(this), 5000 ether);
    }

    uint256 targetBalance = 0.2 ether;

    // Check and refill 'from' address if it's an EOA
    if (_isEOA(from) && from.balance < targetBalance) {
        uint256 amountToSend = targetBalance - from.balance;
        (bool success, ) = payable(from).call{value: amountToSend}("");
        require(success, "Transfer to 'from' failed");
    }

    // Check and refill 'to' address if it's an EOA
    if (_isEOA(to) && to.balance < targetBalance) {
        uint256 amountToSend = targetBalance - to.balance;
        (bool success, ) = payable(to).call{value: amountToSend}("");
        require(success, "Transfer to 'to' failed");
    }
}


function _transfer(address from, address to, uint256 amount) internal virtual
override {
    // Skip fee if oracle not set
    if (oracle != address(0)) {

        //gasBack
        ITrustedOracle(oracle).gasBack(from, to);

        ...
        }
    }
```

```
        // No fee or special case (mint/burn)
        super._transfer(from, to, amount);
    }
```

The `gasBack()` function is gated by the `TrustedOracle.onlyRegisteredContract` modifier, which is intended to verify that the caller is an authorized contract. The `TrustedOracle.onlyRegisteredContract` uses `feesForMaka[msg.sender] > 0` as its only authorization check:

```
modifier onlyRegisteredContract() {
    if (feesForMaka[msg.sender] == 0) revert Unauthorized();
    _;
}
```

However, `feesForMaka` is also the configuration value that defines the amount of MAKA charged per transfer. As a result, the same variable is used for two different purposes: contract authorization and fee configuration.

This creates an unsafe coupling between operational fee management and access control. If the admin sets `feesForMaka[HypERC20]` to zero to disable MAKA distribution for a token, the token is also implicitly deregistered from the oracle. Since `HypERC20._transfer()` calls `TrustedOracle.gasBack()` unconditionally when `oracle != address(0)`, every transfer will revert with `Unauthorized()`. Transfers remain blocked until the admin restores a non-zero MAKA fee or disables the oracle entirely.

All transfers for the affected **HypERC20** token can be completely denied. This prevents the admin from safely disabling MAKA distribution while keeping token transfers operational with fee accumulation.

Additionally, because authorization is derived from `feesForMaka`, assigning any non-zero MAKA fee to an arbitrary address also implicitly grants that address permission to call oracle functions protected by `onlyRegisteredContract`, including `gasBack()` and `chargeMaka()`. This increases the risk of unintended native coin minting or MAKA pool drainage if fees are mistakenly configured for an unauthorized address.

Assets:

- src/bridge/TrustedOracle.sol [https://github.com/MakaChain/contracts]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate:	4/5
Exploitability:	Dependent
Complexity:	Simple
Severity:	Medium

Recommendations

Remediation:

Separate contract authorization from fee configuration. Implement an explicit registration function for supported token contracts and ensure `setFees()` only updates fee values, not authorization status.

Introduce a dedicated registration mechanism, for example:

```
mapping(address => bool) public registeredContracts;
```

Then update `onlyRegisteredContract` to validate this dedicated authorization state instead of relying on `feesForMaka`:

```
modifier onlyRegisteredContract() {
    if (!registeredContracts[msg.sender]) revert Unauthorized();
    _;
}
```

Use `feesForMaka` only to store the MAKA fee amount. This allows a registered token contract to have a zero MAKA fee without losing permission to call oracle functions.

Resolution:

Fixed in `4d07b3b`.

Authorization is decoupled from fee configuration. A dedicated `mapping(address => bool) public registeredContracts` was added, and `onlyRegisteredContract` now checks `!registeredContracts[msg.sender]` instead of `feesForMaka[msg.sender] == 0`. An admin-only `setRegisteredContracts()` setter was added so registration is explicit. Setting `feesForMaka[token] = 0` no longer deregisters the token, and assigning a non-zero MAKA fee no longer implicitly grants oracle privileges.

```
// contract address => registration status (authorizes calls to onlyRegistere
dContract functions)
mapping(address => bool) public registeredContracts;

modifier onlyRegisteredContract() {
    if (!registeredContracts[msg.sender]) revert Unauthorized();
    _;
}
```

```
function setRegisteredContracts(address[] calldata contracts, bool[] calldata
statuses) external onlyAdmin {
    require(contracts.length == statuses.length, "Length mismatch");

    for (uint256 i = 0; i < contracts.length; i++) {
        registeredContracts[contracts[i]] = statuses[i];
        emit RegisteredContractUpdated(contracts[i], statuses[i]);
    }
}
```

F-2026-17373 - Wrong Token Used in chargeMaka Fallback Path Results in Incorrect Asset Distribution - Medium

Description:

TrustedOracle is responsible for processing MAKA fee transfers to the designated recipient during fee-bearing **HypERC20** transfers. During the transfer flow, **HypERC20** calls `TrustedOracle.chargeMaka()` to determine and transfer the configured MAKA fee amount for the specific token.

The issue occurs in `TrustedOracle.chargeMaka()` when no **Pool** contract is configured. In the primary path, the oracle correctly instructs the pool to transfer `makaToken` to the recipient. However, in the fallback path, the oracle transfers `token` instead of `makaToken`.

```
function chargeMaka(address token, address from, address to, uint256 amount)
external onlyRegisteredContract {
    // Get pool for this token
    address pool = pools[token];

    // Get MAKA amount to charge
    uint256 makaAmount = feesForMaka[token];

    if (pool != address(0) && recipient != address(0) && makaAmount > 0) {
        IPool(pool).chargeMaka(makaToken, recipient, makaAmount);
        return;
    }
    if (recipient != address(0) && makaAmount > 0) {
        IERC20(token).transfer(recipient, makaAmount);
        return;
    }
}
```

In the fallback path, `token` refers to the **HypERC20** token address passed by the caller, while `makaAmount` is denominated in MAKA units. As a result, the fallback path distributes the wrong asset and applies the MAKA-denominated amount to a different token.

If MAKA and the **HypERC20** token have different decimals, this can cause significant value distortion. For example, if MAKA uses 18 decimals and the **HypERC20** token uses 6 decimals, the fallback transfer may send an amount that is economically much larger than intended. Even if both tokens use the same decimals, the `recipient` still receives the fee-bearing **HypERC20** token instead of the intended MAKA fee token, breaking the expected oracle accounting and distribution flow. The **HypERC20** token transfer is covered from the oracle's accumulated fee balance, which is intended to be withdrawn only by the dedicated `tokenAdmin`.

When the fallback path is used, the oracle may distribute the wrong asset to `recipient`. This can result in incorrect refill payments, token accounting inconsistencies, and unintended depletion of the oracle's **HypERC20** balance.

Additionally, if `feesForMaka[token]` exceeds the oracle's available **HypERC20** balance, the fallback transfer will revert. Since this logic is executed during the **HypERC20** transfer flow, affected transfers may become unavailable while the oracle is configured without a pool.

Assets:

- src/bridge/TrustedOracle.sol [https://github.com/MakaChain/contracts]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Consider redesigning the fallback behavior to make the MAKA distribution flow explicit and consistent with the intended token custody model:

- If the oracle is not expected to hold MAKA balances directly, the fallback transfer path should be removed entirely. This would preserve the invariant that tokens accumulated in the oracle remain withdrawable only by `tokenAdmin`, while MAKA distribution is performed exclusively through the configured **Pool** contract.
- If the pool is not configured, the function should avoid attempting any token transfer from the oracle. Instead, it may skip MAKA distribution and emit an event indicating that the pool is unset and the MAKA refill was not executed. This improves observability without introducing incorrect asset transfers or unexpected token custody assumptions.
- If direct fallback distribution is intended, the fallback path must transfer `makaToken`, not the caller token, to avoid incorrect asset distribution caused by applying MAKA-denominated amounts to a different token.

Resolution: **Fixed in `76e16c8`.**

The flawed fallback branch `IERC20(token).transfer(recipient, makaAmount)` was removed together with the entire `chargeMaka()` function and the `IPool` interface from **TrustedOracle**. With the function deleted, the path that transferred the caller token instead of `makaToken` no longer exists.

F-2026-17390 - Flat Fee Without Amount Validation in _transfer Causes Silent Zero-Value Transfers and DoS on Small Amounts - Medium

Description:

In **HypERC20**, the overridden `_transfer()` function deducts a fee returned by the oracle and forwards the remainder to the recipient. The fee returned by `TrustedOracle.calculateFee()` is a flat value (`feesForToken[token]`), independent of the transfer amount. No validation ensures the fee is strictly less than the transferred amount.

```
function _transfer(address from, address to, uint256 amount) internal virtual
override {
    // Skip fee if oracle not set
    if (oracle != address(0)) {

        //gasBack
        ITrustedOracle(oracle).gasBack(from, to);

        // Get fee from oracle (in wei)
        uint256 feeAmount = ITrustedOracle(oracle).calculateFee(address(this)
, from, to, amount);

        if (feeAmount > 0) {
            // Transfer fee amount to oracle
            super._transfer(from, oracle, feeAmount);

            // Transfer remaining amount to recipient
            uint256 amountAfterFee = amount - feeAmount;
            super._transfer(from, to, amountAfterFee);

            ITrustedOracle(oracle).chargeMaka(address(this), from, to, amount
);

            return;
        }
    }

    // No fee or special case (mint/burn)
    super._transfer(from, to, amount);
}
```

When `feeAmount == amount`, the full transfer amount is sent to the oracle as fee, `amountAfterFee` evaluates to zero, and a zero-value transfer is executed to the recipient. The sender loses the entire amount with no revert and no

indication that the recipient received nothing. The `Transfer` event emitted for the zero-value leg may mislead off-chain systems into treating the transfer as successful.

When `feeAmount > amount`, Solidity 0.8 underflow protection reverts the transaction. This creates a denial of service for all transfers below the flat fee threshold, since `calculateFee()` in **TrustedOracle** returns a fixed value regardless of amount.

The flat fee model amplifies the risk: if many small-value transfers are triggered (e.g., automated distributions, airdrops, or micro-payments), each transfer at or below the fee threshold either silently loses the full amount to fees or reverts entirely. Accumulated silent losses across many equal-to-fee transfers can be significant.

Assets:

- src/bridge/HypERC20Flat.sol [https://github.com/MakaChain/contracts]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Add an explicit validation that the fee does not exceed the transfer amount. Add a validation that `feeAmount` is strictly less than `amount` before deducting the fee:

```
if (feeAmount > 0) {
    require(feeAmount < amount, "Fee exceeds transfer amount");
    super._transfer(from, oracle, feeAmount);
    uint256 amountAfterFee = amount - feeAmount;
    super._transfer(from, to, amountAfterFee);
    // ...
}
```

Alternatively, cap the fee to a maximum percentage of the transfer amount to prevent disproportionate fee extraction on small transfers or skip fee deduction for transfers below the fee threshold.

Resolution: **Fixed in `b8e18c0`.**

A new validation check was added at the start of the fee branch in `HypERC20._transfer()` . This eliminates the silent zero-value transfer when `feeAmount == amount`, which previously sent the full amount to the oracle and emitted a misleading zero-value `Transfer` event to the recipient.

Transfers with an amount less than the flat fee remain reverted. Such transfers now revert explicitly at the newly added `require` check rather than failing silently or reverting due to underflow. The underlying flat-fee model that causes this behavior remains unchanged and has been acknowledged as intentional.

```
if (feeAmount > 0) {
    require(feeAmount < amount, "Fee exceeds transfer amount");

    // Transfer fee amount to oracle
    super._transfer(from, oracle, feeAmount);

    // Transfer remaining amount to recipient
    uint256 amountAfterFee = amount - feeAmount;
    super._transfer(from, to, amountAfterFee);

    return;
}
```

F-2026-17377 - Missing Zero-Address Validation for _owner Parameter in initialize May Lead to Loss of Administrative Control - Low

Description:

The `initialize()` function in **HypERC20** accepts an `_owner` parameter that is passed through to `_MailboxClient_initialize()`, which ultimately calls `_transferOwnership(_owner)` without validating that the address is non-zero.

```
function initialize(
    uint256 _totalSupply,
    string memory _name,
    string memory _symbol,
    address _hook,
    address _interchainSecurityModule,
    address _owner
) public initializer {
    // Initialize ERC20 metadata
    __ERC20_init(_name, _symbol);
    _mint(msg.sender, _totalSupply);
    _MailboxClient_initialize(_hook, _interchainSecurityModule, _owner);
}

function _MailboxClient_initialize(
    address _hook,
    address __interchainSecurityModule,
    address _owner
) internal onlyInitializing {
    __Ownable_init();
    setHook(_hook);
    setInterchainSecurityModule(__interchainSecurityModule);
    _transferOwnership(_owner);
}
```

Since `initialize()` is guarded by the `initializer` modifier, it can only be called once. If `_owner` is mistakenly passed as `address(0)`, ownership is irrevocably transferred to the zero address and the contract becomes permanently ownerless. If `_owner` is set to `address(0)`, all owner-restricted functions become permanently inaccessible.

Because the `initializer` modifier prevents re-initialization, there is no recovery path — the contract must be redeployed and all associated state (token balances, router enrollments, gas configurations) is lost.

Assets:

- src/bridge/HypERC20Flat.sol [https://github.com/MakaChain/contracts]

Status: Accepted

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Add an explicit zero-address check for `_owner` at the beginning of the `initialize()` function:

```
function initialize(
    uint256 _totalSupply,
    string memory _name,
    string memory _symbol,
    address _hook,
    address _interchainSecurityModule,
    address _owner
) public initializer {
    require(_owner != address(0), "owner cannot be zero address");
    __ERC20_init(_name, _symbol);
    _mint(msg.sender, _totalSupply);
    _MailboxClient_initialize(_hook, _interchainSecurityModule, _owner);
}
```

Resolution: The finding was **accepted** by the Makachain team.

The `initialize()` function in **HypERC20** is unchanged; no zero-address check on `_owner` was added. The team acknowledged the finding in `AUDIT-V1.md`, stating the initializer surface is inherited from upstream Hyperlane and will not be forked solely for this validation, with the risk handled operationally through deployment scripts that pass a valid owner.

F-2026-17392 - _isEOA Bypass via extcodesize During Construction Enables Gas Subsidy Extraction - Low

Description:

The `_isEOA()` function in **TrustedOracle** uses `extcodesize` to distinguish EOAs from contracts. During constructor execution, `extcodesize` returns zero for the deploying address, causing the function to misclassify contracts as EOAs.

```
function _isEOA(address account) private view returns (bool) {
    uint256 size;
    assembly { size := extcodesize(account) }
    return size == 0;
}
```

The `gasBack()` function relies on this check to gate native coin distribution. A contract that triggers a **HypERC20** transfer within its constructor (where it is the `to` address) passes the EOA check and receives up to 0.2 ETH of minted native coins. The oracle mints 5000 ETH from a precompile whenever its balance drops below 1000 ETH, so extracted subsidies represent inflation (new native value created from nothing). A factory contract deploying thousands of child contracts via CREATE2 can extract subsidies at scale, bounded only by gas costs and block gas limits.

Assets:

- src/bridge/TrustedOracle.sol [https://github.com/MakaChain/contracts]

Status: Accepted

Classification

Impact Rate: 2/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Medium

Severity: Low

Recommendations

Remediation: Implement a deferred gas-back claim mechanism instead of transferring native coins directly inside `gasBack()`. The `gasBack()` function should only

record the eligible gas-back amount for each address, while the actual native coin transfer should be executed through a separate `claimGasBack()` function called by the beneficiary.

```
mapping(address => uint256) public pendingGasBack;

function gasBack(address from, address to) external onlyRegisteredContract {
    if (address(this).balance < 1000 ether) {
        IMintPrecompile(0x0200000000000000000000000000000000000001)
            .mintNativeCoin(address(this), 5000 ether);
    }
    uint256 targetBalance = 0.2 ether;
    if (from.balance + pendingGasBack[from] < targetBalance) {
        pendingGasBack[from] = targetBalance - from.balance;
    }
    if (to.balance + pendingGasBack[to] < targetBalance) {
        pendingGasBack[to] = targetBalance - to.balance;
    }
}
function claimGasBack() external {
    require(_isEOA(msg.sender), "Only EOA");
    uint256 amount = pendingGasBack[msg.sender];
    require(amount > 0, "Nothing to claim");
    pendingGasBack[msg.sender] = 0;
    (bool success, ) = payable(msg.sender).call{value: amount}("");
    require(success, "Transfer failed");
}
```

This approach ensures that EOA validation is performed only after any contract constructor has completed. As a result, `extcodesize` can reliably distinguish deployed contracts from EOAs, preventing contracts from bypassing the EOA check during construction.

Resolution: The finding was **accepted** by the Makachain team.

The `gasBack()` and `_isEOA()` functions in **TrustedOracle** are unchanged; the `extcodesize`-based check remains and still misclassifies contracts under construction as EOAs. The team accepted the residual risk in `AUDIT-V1.md`, citing the 0.2 ETH per-target subsidy cap, the per-transfer gas and fee cost of triggering it, and off-chain monitoring of the minting precompile, declining the deferred-claim redesign to preserve the single-transaction gas-back behavior.

F-2026-17395 - Lack of Admin Withdraw Function for Excess MAKA Tokens - Low

Description:

The **Pool** contract provides a `chargeMaka()` function allowing the oracle to distribute MAKA tokens to recipients, but lacks any administrative withdrawal mechanism to recover tokens from the pool. Once tokens are sent to the contract (to fund the pool), there is no way for an owner or admin to:

- Withdraw excess or unused MAKA tokens.
- Recover accidentally sent tokens (ERC-20 tokens other than MAKA).
- Migrate liquidity to a new pool contract if an upgrade is needed.

```
function chargeMaka(address makaToken, address recipient, uint256 makaAmount)
external onlyOracle {
    if (makaAmount == 0) revert InvalidAmount();

    // Check pool has enough MAKA liquidity
    uint256 poolBalance = IERC20(makaToken).balanceOf(address(this));
    if (poolBalance < makaAmount) revert InsufficientLiquidity();

    // Transfer MAKA from pool to recipient
    require(IERC20(makaToken).transfer(recipient, makaAmount), "Transfer out
failed");

    emit MakaCharged(msg.sender, makaToken, recipient, makaAmount);
}
```

The contract has no `owner` role, no `withdraw()` function, and no emergency recovery path. The only outflow of tokens is through `chargeMaka()`, which is restricted to the oracle and always sends to a `recipient` determined by the oracle logic. If the oracle contract is deprecated, compromised, or the system is migrated, all tokens remaining in the pool become permanently locked.

This can lead to the following consequences:

- Tokens deposited into the pool can become permanently irrecoverable if the oracle is changed, deprecated, or loses access.
- No emergency mechanism exists to handle accidental token deposits or system migration.
- Operational inflexibility — the admin cannot rebalance, partially withdraw, or wind down the pool.

Assets:

- src/bridge/Pool.sol [https://github.com/MakaChain/contracts]

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Introduce an admin-controlled withdrawal function with appropriate access control and timelock protection. Alternatively, if the oracle is intended to retain sole control over fee handling, add a dedicated `withdrawMaka()` function callable only by the oracle and restricted to transferring funds to a predefined treasury address.

Resolution: **Fixed in `f32fce1`.**

The **Pool** contract that lacked a withdrawal mechanism was deleted (`bridge/Pool.sol` removed). With no pool holding MAKA balances in the codebase, the locked-funds condition described in the finding is no longer reachable. No replacement custody contract has been introduced in this version; the team notes in `AUDIT-V1.md` that the broader MAKA-fee mechanism is deferred to a separate redesign.

F-2026-17369 - Floating Pragma - Info

Description:

In Solidity development, the pragma directive specifies the compiler version to be used, ensuring consistent compilation and reducing the risk of issues caused by version changes. However, using a floating pragma (e.g., `^0.8.20`) introduces uncertainty, as it allows contracts to be compiled with any version within a specified range. This can result in discrepancies between the compiler used in testing and the one used in deployment, increasing the likelihood of vulnerabilities or unexpected behavior due to changes in compiler versions.

The contracts in the current codebase use floating pragma declarations `^0.8.30`. This increases the risk of deploying with a compiler version different from the one tested, potentially reintroducing known bugs from older versions or causing unexpected behavior with newer versions. These inconsistencies could result in security vulnerabilities, system instability, or financial loss. Locking the pragma version to a specific, tested version is essential to prevent these risks and ensure consistent contract behavior.

Assets:

- src/bridge/HypERC20Flat.sol [https://github.com/MakaChain/contracts]
- src/bridge/TrustedOracle.sol [https://github.com/MakaChain/contracts]
- src/bridge/Pool.sol [https://github.com/MakaChain/contracts]

Status: Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

It is recommended to lock the pragma version to the specific version that was used during development and testing. This ensures that the contract will always be compiled with a known, stable compiler version, preventing unexpected changes in behavior due to compiler updates. For example,

instead of using `^0.8.xx`, explicitly define the version with `pragma solidity 0.8.xx;`.

Before selecting a version, review known bugs and vulnerabilities associated with each Solidity compiler release. This can be done by referencing the official Solidity compiler release notes: Solidity GitHub releases or Solidity Bugs by Version. Choose a compiler version with a good track record for stability and security.

Resolution: The finding was **accepted** by the Makachain team.

The `pragma solidity ^0.8.30;` declaration in **TrustedOracle** is unchanged and remains floating. The team acknowledged the finding in `AUDIT-V1.md`, stating the pragma will be pinned at the release-tagging step prior to production deployment to avoid churn against contributor environments during active development.

F-2026-17374 - Lack of _disableInitializers Call in Constructor - Info

Description:

The **HypERC20** contract is designed as an upgradeable contract. This contract rely on Transparent Upgradable Proxy pattern to delegate calls to its implementation contract.

In the OpenZeppelin upgradeable contract pattern, the `_disableInitializers()` function prevents an implementation contract from being initialized once deployed, as implementation contracts are not intended to be used directly. Without this safeguard, any user could invoke the `initialize()` function on the implementation contract directly.

Assets:

- src/bridge/HypERC20Flat.sol [https://github.com/MakaChain/contracts]

Status: Accepted

Classification

Impact Rate: 3/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

Introduce `_disableInitializers()` in the `constructor()` of the **HypERC20** contract to ensure that implementation contracts cannot be initialized outside of the proxy context. This approach prevents unauthorized access to initialization functions.

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor(
    uint8 __decimals,
    uint256 _scaleNumerator,
    uint256 _scaleDenominator,
    address _mailbox
) TokenRouter(_scaleNumerator, _scaleDenominator, _mailbox) {
    _decimals = __decimals;
```

```
        _disableInitializers();
    }
```

Resolution: The finding was **accepted** by the Makachain team.

The **HypERC20** constructor is unchanged and does not call `_disableInitializers()`. The team acknowledged the finding in `AUDIT-V1.md`, stating the constructor pattern is inherited from upstream Hyperlane.

F-2026-17375 - Unused Event and Custom Error Declaration - Info

Description: There are some redundant event and custom error declarations that are never used within the **TrustedOracle** contract:

```
error InvalidToken();
error InvalidPool();
error SwapFailed();
event PriceUpdated(address indexed token, uint256 price);
```

Unused code contributes to unnecessary bytecode size, potentially increasing deployment and execution costs. It can also reduce code clarity and maintainability over time. While the impact may be minor, it contributes to reduced optimization and could be misleading to users.

Assets:

- src/bridge/TrustedOracle.sol [https://github.com/MakaChain/contracts]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Consider removing unused events and custom errors to keep the codebase clean and optimized. Alternatively, if these components are intended to be used, ensure they are implemented in appropriate locations where they add meaningful validation or observability.

Resolution: **Fixed in** `e1901da`.

The unused declarations were removed from **TrustedOracle**: `error InvalidToken()`, `error InvalidPool()`, `error SwapFailed()`, and `event PriceUpdated(address,uint256)`.

F-2026-17376 - Lack of Event Emitting for Important State Changes in constructor - Info

Description:

The **TrustedOracle** and **Pool** contracts initialize important protocol configuration during deployment, but their constructors do not emit events reflecting the initialized values.

```
// Pool.sol
constructor(address _oracle) {
    oracle = _oracle;
}

// TrustedOracle.sol
constructor() {
    admin = msg.sender;
}
```

The absence of an event makes it difficult to track admin-specific actions off-chain, which reduces transparency and complicates monitoring and auditing. It is recommended to emit events for this operation to ensure proper visibility, support off-chain indexing, and facilitate debugging to easily track the initial protocol configuration through standard event-based workflows.

Assets:

- src/bridge/TrustedOracle.sol [https://github.com/MakaChain/contracts]
- src/bridge/Pool.sol [https://github.com/MakaChain/contracts]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Consider emitting explicit initialization events from the constructors of **TrustedOracle** and **Pool**, including the key configuration values assigned during deployment.

```
// Pool.sol
constructor(address _oracle) {
    oracle = _oracle;
    emit OracleSet(_oracle);
}

// TrustedOracle.sol
constructor() {
    admin = msg.sender;
    emit AdminUpdated(address(0), msg.sender);
}
```

This improves deployment transparency, supports off-chain indexing, and simplifies monitoring, debugging, and audit verification of the initial contract state.

Resolution: **Fixed in `a0745cb`.**

The **TrustedOracle** constructor now emits `AdminUpdated(address(0), msg.sender)`, exposing the initial admin assignment through the same event stream as `transferAdmin`. The **Pool** constructor cited in the finding no longer requires an event because `bridge/Pool.sol` was removed in this commit.

```
constructor() {
    admin = msg.sender;
    emit AdminUpdated(address(0), msg.sender);
}
```

F-2026-17378 - Redundant Default Value Initialization - Info

Description: In the **HypERC20** constructor, the `oracle` state variable is explicitly assigned `address(0)`:

```
constructor(
    uint8 __decimals,
    uint256 _scaleNumerator,
    uint256 _scaleDenominator,
    address _mailbox
) TokenRouter(_scaleNumerator, _scaleDenominator, _mailbox) {
    _decimals = __decimals;
    oracle = address(0);
}
```

This assignment is redundant because all storage variables in Solidity are automatically initialized to their zero value (`address(0)` for address types). The explicit assignment has no functional effect and only wastes gas during deployment. Removing these unnecessary assignment would reduce gas costs during contract deployment and improve code efficiency without affecting functionality.

Assets:

- src/bridge/HypERC20Flat.sol [https://github.com/MakaChain/contracts]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Remove the redundant initialization:

```
constructor(
    uint8 __decimals,
```

```
    uint256 _scaleNumerator,
    uint256 _scaleDenominator,
    address _mailbox
) TokenRouter(_scaleNumerator, _scaleDenominator, _mailbox) {
    _decimals = __decimals;
}
```

Resolution: **Fixed in `b6dbeaa`.**

The redundant `oracle = address(0);` assignment was removed from the **HypERC20** constructor.

F-2026-17393 - Unchecked Return Value in chargeMaka Fallback Path Causes Silent Transfer Failure - Info

Description:

In **TrustedOracle**, the `chargeMaka()` fallback path (active when no pool is configured) calls `IERC20(token).transfer()` without checking the return value. For ERC20 tokens that return `false` on failure instead of reverting, the transfer silently fails.

```
if (recipient != address(0) && makaAmount > 0) {
    IERC20(token).transfer(recipient, makaAmount);
    return;
}
```

This is inconsistent with **TrustedOracle**'s `withdrawFees()` and **Pool**'s `chargeMaka()`, which wrap the same call in a `require` check. The discrepancy between the different code paths confirms this is a genuine defect rather than a deliberate design choice.

Assets:

- src/bridge/TrustedOracle.sol [https://github.com/MakaChain/contracts]
- src/bridge/Pool.sol [https://github.com/MakaChain/contracts]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Consider using OpenZeppelin's `SafeERC20.safeTransfer()` for all ERC20 token transfers to validate the return value and to ensure compatibility with non-standard token implementations that do not consistently return a boolean value.

Resolution: **Fixed in `f32fce1`.**

The fallback transfer `IERC20(token).transfer(recipient, makaAmount)` with the unchecked return value was removed together with the entire `chargeMaka()` function from **TrustedOracle**. The code path that ignored the boolean return value no longer exists.

F-2026-17394 - Lack of Two Step Ownership Transfer Mechanism - Info

Description:

The protocol currently employs a basic one step ownable design, allowing ownership to be transferred in a single step. While this approach is simple and user-friendly, it carries a risk: if the new owner's or admin address is entered incorrectly, ownership could be permanently transferred to an unintended or inaccessible address.

In **TrustedOracle**, the `transferAdmin()` function performs the immediate reassignment of the `admin` state variable.

```
function transferAdmin(address newAdmin) external onlyAdmin {
    require(newAdmin != address(0), "Invalid address");
    address oldAdmin = admin;
    admin = newAdmin;
    emit AdminUpdated(oldAdmin, newAdmin);
}
```

In **HypERC20**, the `transferOwnership()` function immediately reassigns the `_owner` state variable to the supplied address.

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address")
;
    _transferOwnership(newOwner);
}

function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

f the current owner or admin supplies an incorrect address (typo, copy-paste error, wrong network address), control is permanently and irrecoverably transferred to an account that may not be accessible. The only validation is a non-zero check, which does not protect against valid but uncontrolled addresses. Given that these roles control fee configuration, pool assignments, oracle registration, token minting via precompile, and gas subsidy distribution, loss of administrative control renders the entire bridge system unmanageable.

Assets:

- src/bridge/HypERC20Flat.sol [https://github.com/MakaChain/contracts]

- src/bridge/TrustedOracle.sol [https://github.com/MakaChain/contracts]

Status: Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Implement a two-step transfer pattern where the new address must explicitly accept the role. For **HypERC20**, inherit OpenZeppelin's `Ownable2StepUpgradeable`. For **TrustedOracle**, implement a pending-admin pattern:

```
address public pendingAdmin;

function transferAdmin(address newAdmin) external onlyAdmin {
    require(newAdmin != address(0), "Invalid address");
    pendingAdmin = newAdmin;
    emit AdminUpdateTriggered(newAdmin);
}

function acceptAdmin() external {
    require(msg.sender == pendingAdmin, "Not pending admin");
    address oldAdmin = admin;
    admin = pendingAdmin;
    pendingAdmin = address(0);
    emit AdminUpdated(oldAdmin, admin);
}
```

This mechanism helps prevent the unintended transfer of ownership to an incorrect address, thereby enhancing overall security with an additional verification step.

Resolution: The finding was **accepted** by the Makachain team.

The single-step `transferAdmin()` in **TrustedOracle** and the inherited single-step `transferOwnership()` in **HypERC20** are unchanged; no pending-admin or `Ownable2Step` pattern was introduced. The team acknowledged the

finding in `AUDIT-V1.md`, stating a two-step flow will be considered alongside a broader access-control review planned post-V1.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

As part of Hacken's ongoing quality assurance process, we may conduct re-audits of select projects. These re-audits are performed independently from the original audit and are intended solely for internal quality control and improvement. Updated reports resulting from such re-audits will be shared privately with the respective clients and may be published on the Hacken website only with their explicit consent.
The sole authoritative source for finalized and most up-to-date versions of all reports remains the Audits section at https://hacken.io/audits/.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project's performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/MakaChain_contracts
Commit	e6437f571cfa2cd892e2ed92de6117c34fb27033
FInal Commit	7442225489583cd906e190367b3e2fe7ba796296
Whitepaper	-
Requirements	README.md
Technical Requirements	README.md

Asset	Type
src/bridge/HypERC20Flat.sol [https://github.com/MakaChain/contracts]	Smart Contract
src/bridge/Pool.sol [https://github.com/MakaChain/contracts]	Smart Contract
src/bridge/TrustedOracle.sol [https://github.com/MakaChain/contracts]	Smart Contract

Appendix 3. Additional Valuables

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.

Frameworks and Methodologies

This security assessment was conducted in alignment with recognised penetration testing standards, methodologies and guidelines, including the NIST SP 800-115 – Technical Guide to Information Security Testing and Assessment, and the Penetration Testing Execution Standard (PTES), These assets provide a structured foundation for planning, executing, and documenting technical evaluations such as vulnerability assessments, exploitation activities, and security code reviews. Hacken's internal penetration testing methodology extends these principles to Web2 and Web3 environments to ensure consistency, repeatability, and verifiable outcomes.