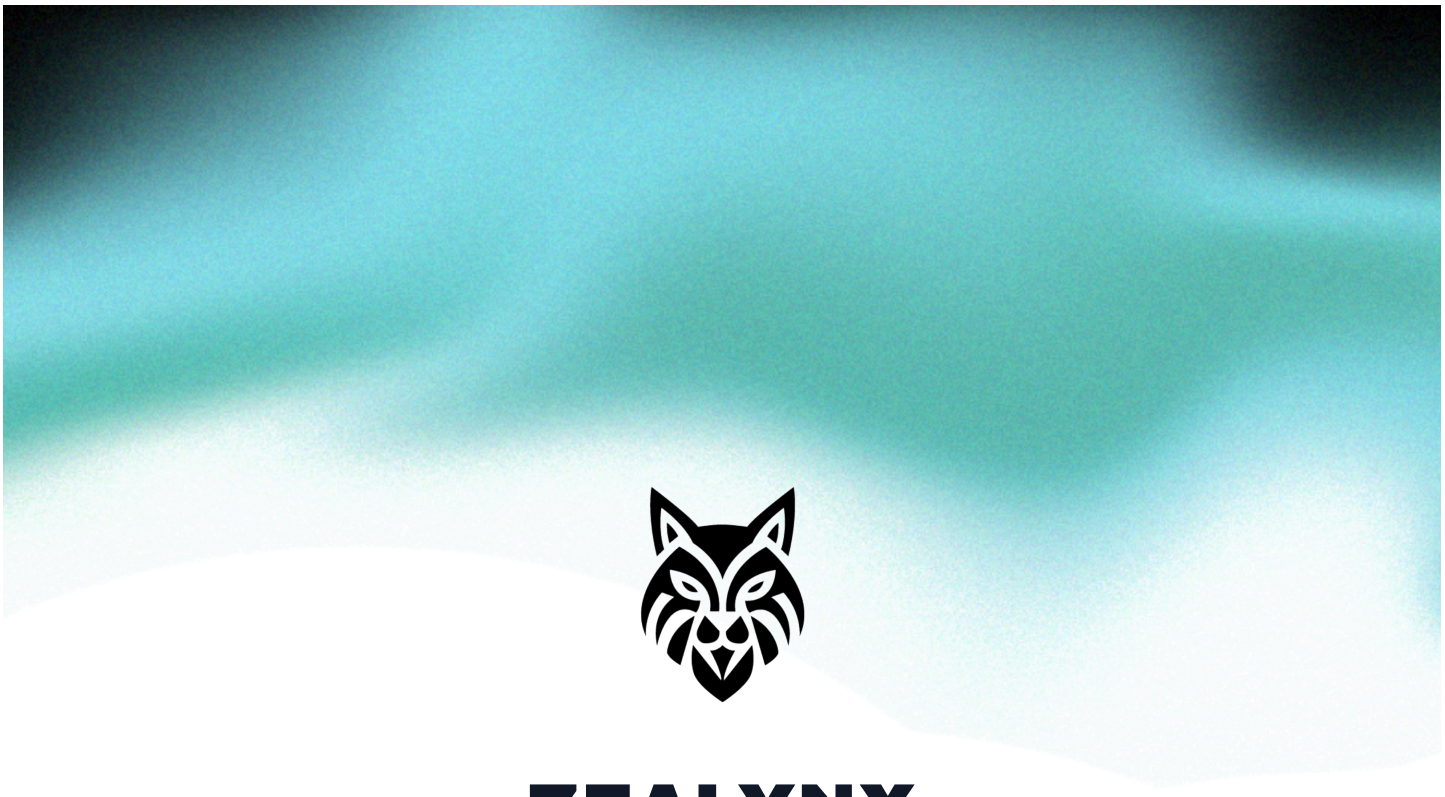

ZEALYNX

Web3 Security & Smart Contract Development

YadaCoin

Smart Contract Audit

05 March 2026

Zealynx

contact@zealynx.io

Carlos (Bloqarl)	@TheBlockChainer
Bobby_Petrov	@0x_bob_0x
derastephh	@derastephh

Contents

1. About Zealynx

Zealynx, founded in January 2024 by Carlos (Bloqarl), specializes in smart contract audits, and development. Our services include comprehensive smart contract audits, application security audits, such as pentesting, and AI Audits. We are trusted by clients such as Badger DAO, Ample protocol, Lido, Inverter, Matchain, and Golden Grid.

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our website Zealynx.io and our blogs.

Zealynx has achieved public recognition, including winning 1st prize in the Uniswap Hook Hackathon and a Top 5 position in the Beanstalk Audit public contest. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can't guarantee there are no vulnerabilities. It's a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

Zealynx Security conducted a security audit of the YadaCoin bridge infrastructure — a cross-chain bridge between the BNB and YadaCoin blockchains, and an on-chain bridge between BNB/ERC20 tokens and secure wrapped tokens. The audit covered the core Bridge contract, key pair registration system, wrapped token factory and proxy architecture, and associated ERC-20 mock contracts.

3.2 About YadaCoin

YadaCoin is a blockchain project with a cross-chain bridge that enables asset transfers between BNB and YadaCoin blockchain. The bridge implements a KERI-based key pair registration system (KeyLogRegistry), native and ERC-20 token wrapping/unwrapping via an upgradeable beacon proxy architecture, and an ERC-2612 permit-based operation flow.

3.3 Audit Scope

The code under review is composed of multiple smart contracts written in Solidity and includes ~1,159 nSLOC — normalized source lines of code. The codebase implements a cross-chain bridge architecture covering key pair registration, native and ERC-20 token wrapping/unwrapping, an upgradeable beacon proxy system, and permit-based batch operation flows.

- Bridge.sol
- KeyLogRegistry.sol
- MockERC20.sol
- WrappedToken.sol
- WrappedTokenBeacon.sol
- WrappedTokenFactory.sol
- WrappedTokenProxy.sol

Repository: https://github.com/pdxwebdev/yadakeyeventwallet
Commit: 72bee2627de93306e59f3358a33336ae08cd7daa

4. Audit Methodology

Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough **Fuzz** and **Invariant Testing**, and **meticulous manual security review.**

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

1. **Code Quality:** We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
2. **Best Practices:** Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
3. **Documentation and Comments:** We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

6.1 The Key Findings

Over a 1-week engagement, the Zealynx team conducted a security review of the YadaCoin bridge infrastructure — a cross-chain bridge enabling asset transfers between BNB Chain and YadaCoin Blockchain, focusing on the bridge's key pair registration system, native and ERC-20 token wrapping/unwrapping mechanisms, permit-based operation flow, and the upgradeable proxy architecture. A total of 28 issues were identified: 3 Critical, 2 High, 4 Medium, 16 Low, and 3 Informational.

Zero-cost bridge drain via missing msg.value validation: The `transferOnly` remainder path in the Bridge contract does not validate `msg.value`, allowing any caller to drain the entire native balance of the bridge at zero cost. This is an immediate, exploitable critical vulnerability requiring urgent remediation.

Complete key pair registration failure post-upgrade: The `confirmingPublicKeyHash` is derived from the wrong public key, causing every key pair registration to revert after the inception key. This effectively locks all users out of the bridge's core onboarding flow and must be fixed before any mainnet deployment.

New user onboarding regression in registerKeyLog: A regression introduced during an upgrade broke the `registerKeyLog` function for new users, preventing them from onboarding entirely. Combined with the above, the bridge cannot accept new participants in its current state.

Unbacked wrapped token minting via msg.value reuse: In permit-based batch operations, the same `msg.value` is accepted as valid collateral for multiple native wrap permits within a single transaction. An attacker can mint an arbitrary amount of wrapped tokens while only paying once, enabling unbounded inflation of the wrapped token supply.

While the bridge architecture demonstrates thoughtful design — including upgradeable proxies, permit-based flows, and a KERI-inspired key registration system — the critical and high severity findings represent fundamental correctness issues that must be resolved before the protocol can safely handle user funds.

6.2 Architectural Security Observations

During the assessment, we observed several architectural decisions and controls that contribute to the YadaCoin Bridge's security posture:

KERI-Based Key Event Chain Integrity: The `KeyLogRegistry` enforces strict chain integrity across key events, with uniqueness enforcement across all four index mappings (`publicKeyHashToIndex`, `prerotatedKeyHashToIndex`, `transactionHashToIndex`, and `confirmedToIndex`). This prevents replay, reordering, and duplication of key events.

Dual-Signature Key Pair Registration: Key pair registration requires both the unconfirmed and confirming keys to produce valid signatures with paired nonces. This two-party verification model ensures that no single compromised key can unilaterally register a new key pair.

Atomic State Transitions: Functions like `registerKeyPairWithTransfer` combine key registration, token pair setup, permit execution, and ownership transfer in a single transaction. This eliminates intermediate inconsistent states that could be exploited between separate calls.

Key-Rotation-Gated Upgrades: The UUPS upgrade path (`upgradeWithKeyRotation`) is tied to the key rotation system rather than a simple owner call. This raises the bar for unauthorized contract upgrades beyond standard `onlyOwner` patterns.

Beacon Proxy Pattern for Wrapped Tokens: Wrapped tokens are deployed via `UpgradeableBeacon`, allowing the implementation to be upgraded uniformly across all deployed wrapped tokens without requiring individual proxy upgrades.

Defense-in-Depth on External Calls: State-changing functions that interact with external tokens use `SafeERC20` for transfers, `ReentrancyGuardUpgradeable` on critical paths, and the checks-effects-interactions pattern consistently.

6.3 Security Strengths Observed

Based on our assessment, YadaCoin demonstrates the following security strengths:

Sound Core Key Management Design: The KERI-inspired key event log — the protocol's core value proposition — is well-engineered. Sequence validation, hash uniqueness, event flag enforcement, and prerotated key commitments all function correctly. No critical vulnerabilities were found in this subsystem.

Modern Solidity Practices: The codebase consistently uses custom errors over `require` strings, `SafeERC20` for token interactions, upgradeable contract patterns with initializers, and explicit reentrancy guards — reflecting current best practices.

Findings Concentrated in Edge Cases: The majority of identified issues relate to input validation gaps, inconsistent error handling, and edge cases around non-18-decimal tokens and native token handling — not to the core key management or bridge logic where the highest-risk attack surface exists.

Controlled Privilege Model: Ownership transfer is gated by the `KeyLogRegistry` validation rather than a simple `transferOwnership` call. This significantly reduces the risk of unauthorized ownership changes compared to standard `Ownable` patterns.

Clear Separation of Concerns: The architecture cleanly separates key management (`KeyLogRegistry`), bridge operations (`Bridge`), and token logic (`WrappedToken`), with well-defined interfaces between components. This modular design limits the blast radius of bugs in any single component.

Summary of Findings:

Vulnerability	Severity	Status
[C-1] New Users Cannot Onboard After Upgrade Due to registerKeyLog Regression	Critical	Fixed
[C-2] confirmingPublicKeyHash derived from wrong public key causes all key pair registrations to revert after inception	Critical	Fixed
[C-3] Missing msg.value validation in transferOnly remainder path enables zero-cost drain of entire bridge native balance	Critical	Fixed
[H-1] Transaction-level msg.value reused across multiple native wrap permits enables minting unbacked wrapped tokens	High	Fixed
[H-2] Unvalidated user-provided wrapped token address in pair registration enables binding legitimate tokens to malicious contracts	High	Fixed
[M-1] ERC-2612 permit frontrunning allows griefing DoS on all permit-dependent Bridge operations	Medium	Fixed
[M-2] `rotateToPublicKey` only transfers Bridge ownership leads to permanent KeyLogRegistry desynchronization and retained privileged access by old owner	Medium	Fixed
[M-3] Forced feeCollector update during ownership transfer leads to potential fee misdirection	Medium	Fixed
[M-4] Gas DoS via Key Chain Traversal	Medium	Fixed
[L-1] No refund of excess `msg.value` in native BNB operations leads to permanent absorption of overpaid funds	Low	Fixed
[L-2] Absence of emergency pause mechanism across all contracts leads to inability to halt operations during active exploits	Low	Fixed
[L-3] Hardcoded 18-Decimal Wrapped Tokens Break Unwrap Fees and Wallet Displays	Low	Fixed

Vulnerability	Severity	Status
[L-4] No mechanism to deregister or disable token pairs leads to permanent persistence of compromised or deprecated pairs	Low	Fixed
[L-5] Silent chain truncation at 1000 entries causes `getLatestChainEntry` to return stale data with no error indication	Low	Fixed
[L-6] Missing `_disableInitializers()` in constructor allows initialization of implementation contracts	Low	Fixed
[L-7] Early loop break in `_executePermits` causes order-dependent revert on valid owner transactions	Low	Fixed
[L-8] `_handleMint` Fee Is Computed but Never Collected	Low	Fixed
[L-9] Mandatory Native Permit Check Bypassable with Zero Value	Low	Fixed
[L-10] Broken Logic in `Bridge::getLatestEntryByPrerotatedKeyHash`	Low	Fixed
[L-11] Owner Can Drain Bridge via emergencyWithdrawBNB	Low	Fixed
[L-12] feeCollector Silently Overwritten by Arbitrary User	Low	Fixed
[L-13] Fixed 30,000 gas stipend in `_transferNative` leads to potential incompatibility with contract recipients	Low	Fixed
[L-14] Missing enforcement of `Bridge::MAX_TOKEN_PAIRS`	Low	Fixed
[L-15] Missing fee-on-transfer accounting in `_handleWrap` leads to potential under-collateralization of wrapped tokens	Low	Fixed
[L-16] Extra Argument in `WrappedToken` Initialization is Ignored	Low	Fixed
[I-1] Duplicate Public Key Length Validation in `Bridge::registerKeyPairWithTransfer`	Info	Fixed

Vulnerability	Severity	Status
[I-2] Redundant Zero Address Check in `Bridge::registerKeyPairWithTransfer` and `Bridge::rotateToPublicKey`	Info	Fixed
[I-3] Dead Code in `_handleWrap` and `_handleUnwrap`	Info	Fixed

7. Audit Findings

7.1 Critical Severity Findings

[C-01] New Users Cannot Onboard After Upgrade Due to registerKeyLog Regression

Affected files

`KeyLogRegistryUpgrade.sol` lines 64-78, 177-184

Description

Note: The problem described in `[C-02]` is fixed in `KeyLogRegistryUpgrade.sol` but it introduce a different type of regression described here. KeyLogRegistryUpgrade.sol and KeyLogRegistry.sol should not be used in production before the recommended fix described below.

The `[C-02]` fix in `KeyLogRegistryUpgrade` changed line 184 of `validateTransaction` from:

```
// Original (buggy):
address confirmingPublicKeyHash =
getAddressFromPublicKey(unconfirmed.publicKey);

// Upgrade (fixed for pairs, but breaks single
registration):
address confirmingPublicKeyHash =
getAddressFromPublicKey(confirming.publicKey);
```

This fix is correct for `registerKeyLogPair` (where `confirming.publicKey` is a real 64-byte key). However, `registerKeyLog` passes an empty confirming key:

```
function registerKeyLog(KeyData memory key) external
onlyAuthorized {
    address publicKeyHash =
getAddressFromPublicKey(key.publicKey);
    (KeyEventFlag flag, ) = validateTransaction(
        key,
        KeyData({
            publicKey: "",           // <-- empty bytes
            prerotatedKeyHash: address(0),
            twicePrerotatedKeyHash: address(0),
            outputAddress: address(0),
            prevPublicKeyHash: address(0)
        }),
        false                        // isPair = false
    );
```

`validateTransaction` calls `getAddressFromPublicKey(confirming.publicKey)` at line 184 **unconditionally** — before the `if (!isPair) return` guard at line 211. Since `confirming.publicKey` is `""` (0 bytes), `getAddressFromPublicKey` reverts with `"Public key must be 64 bytes"`.

This means `registerKeyLog` **always reverts** after upgrading to `KeyLogRegistryUpgrade`.

Vulnerable Scenario:

The following steps help understand the issue:

1. Protocol deploys `KeyLogRegistryUpgrade`
2. New user calls `Bridge.registerKeyPairWithTransfer()` with no confirming key (`ctx.confirming.outputAddress == address(0)`)
3. Bridge evaluates `isPair = false`
4. Bridge calls `keyLogRegistry.registerKeyLog(unconfirmedKeyData)`
5. `registerKeyLog` constructs an empty confirming `KeyData` with `publicKey: ""` and calls `validateTransaction(key, emptyConfirming, false)`
6. `validateTransaction` executes:

`getAddressFromPublicKey(confirming.publicKey)` — this is `getAddressFromPublicKey("")`
7. `getAddressFromPublicKey` checks `require(publicKey.length == 64)` — fails because length is 0
8. Transaction reverts with `"Public key must be 64 bytes"` — inception never registered
9. Result: no new user can register their first key on the bridge

Impact

After deploying KeyLogRegistryUpgrade:
- **All new user onboarding breaks** — no inception events can be registered
- Existing users with confirmed key pairs can still rotate (via `registerKeyLogPair`)
- The protocol cannot onboard any new users until a second upgrade fixes this regression

Recommendations

Move the `getAddressFromPublicKey(confirming.publicKey)` call inside the `isPair` block:

```
function validateTransaction(
    KeyData memory unconfirmed,
    KeyData memory confirming,
    bool isPair
) public view returns (KeyEventFlag, KeyEventFlag) {
    (KeyLogEntry memory lastEntry, bool hasEntries) =
getLatestChainEntry(unconfirmed.publicKey);
    address unconfirmedPublicKeyHash =
getAddressFromPublicKey(unconfirmed.publicKey);
-   address confirmingPublicKeyHash =
getAddressFromPublicKey(confirming.publicKey);
+   address confirmingPublicKeyHash;  // only computed
```

```
for pairs

    // ... existing validation for unconfirmed ...

    if (!isPair) {
        require(unconfirmed.prerotatedKeyHash ==
unconfirmed.outputAddress, "...");
        require(confirming.outputAddress == address(0),
"...");
        return (firstFlag, KeyEventFlag.UNCONFIRMED);
    }

+   confirmingPublicKeyHash =
getAddressFromPublicKey(confirming.publicKey);
    // ... rest of pair validation ...
```

YadaCoin

Confirmed. Moved `confirmingPublicKeyHash` computation inside the `isPair` block so that non-pair `registerKeyLog` calls no longer revert on the missing confirming public key.

Zealynx

Fixed. Verified the fix resolves the regression for non-pair registrations. Also identified and confirmed removal of a redundant `require` statement that was unreachable after the restructuring.

[C-02] confirmingPublicKeyHash derived from wrong public key causes all key pair registrations to revert after inception

Affected files

`KeyLogRegistry.sol` line 184

Description

NOTE: this bug is fixed in KeyLogRegistryUpgrade.sol, and KeyLogRegistry.sol should not be used in production!

In `KeyLogRegistry.sol`, `validateTransaction` computes the confirming key's hash from the **wrong** public key:

```
// KeyLogRegistry.sol line 183-184
address unconfirmedPublicKeyHash =
getAddressFromPublicKey(unconfirmed.publicKey);
address confirmingPublicKeyHash =
getAddressFromPublicKey(unconfirmed.publicKey); // BUG:
should be confirming.publicKey
```

Both hashes are derived from `unconfirmed.publicKey`. The subsequent check at line 217:

```
require(getAddressFromPublicKey(confirming.publicKey) ==
confirmingPublicKeyHash, "Invalid confirmingPublicKey");
```

Requires `hash(confirming.publicKey) == hash(unconfirmed.publicKey)`, which is always false for valid key pairs.

Vulnerable Scenario:
The following steps help understand the issue:
1. Deploy `KeyLogRegistry` behind a UUPS proxy.
2. Register key A via `registerKeyLog` (inception) — **succeeds** because `isPair=false` returns at line 214 before the buggy check at line 217 is reached.
3. Attempt key rotation by calling `registerKeyLogPair(keyB, keyC)` where:
- Key B is the next rotation key (`prevPublicKeyHash = hash(A)`)
- Key C is the confirming key (`prevPublicKeyHash = hash(B)`)
4. `validateTransaction` computes:
- `unconfirmedPublicKeyHash = hash(B)` (line 183 — correct)
- `confirmingPublicKeyHash = hash(B)` (line 184 — **BUG**, should be `hash(C)`)
5. Line 217 evaluates: `require(hash(C) == hash(B))` — **always false**, reverts with "Invalid confirmingPublicKey".
6. Key chain is permanently stuck at inception. No rotation, wrap, or unwrap is possible.

Impact

All `registerKeyLogPair` calls revert with "Invalid confirmingPublicKey"

No key rotation pairs can be registered — key chains are permanently stuck after inception

All wrap/unwrap operations that require pair registration are blocked

Bridge is non-functional for any operation requiring key pair rotation

Recommendations

Fixed in `KeyLogRegistryUpgrade.sol` line 184:

```
address confirmingPublicKeyHash =
getAddressFromPublicKey(confirming.publicKey); // FIXED
```

YadaCoin

Confirmed.

Zealynx

Fixed.

[C-03] Missing msg.value validation in transferOnly remainder path enables zero-cost drain of entire bridge native balance

Affected files

`Bridge.sol` lines 331-341

Description

The `_executePermits` function processes native tokens transfers without validating that `msg.value` covers the total transfer amount. For the `transferOnly` code path, the bridge sends native token from its own reserves based solely on `permit.amount` with **zero validation** that the caller provided sufficient `msg.value`.

Root Cause
For ERC20 tokens, the `permit + safeTransferFrom` mechanism naturally limits transfers to what the user signed for. For native tokens (`address(0)`), no equivalent mechanism exists. The bridge simply calls `_transferNative` using its own balance.

```
// Bridge.sol lines 331-341
if (transferOnly) {
    uint256 remainder = permit.amount - totalTransferred;
    if (remainder > 0) {
        if (ectx.token == address(0)) {
            _transferNative(ectx.prerotatedKeyHash,
remainder); // Sends from bridge reserves!
        } else {

IERC20(permit.token).safeTransferFrom(ectx.user,
ectx.prerotatedKeyHash, remainder);
        }
        totalTransferred += remainder;
```

```
    }
}
```

No check exists that `msg.value >= permit.amount` or `msg.value >= totalTransferred`.

Vulnerable Scenario:

Prerequisites: Bridge holds any amount of native tokens from legitimate users' wraps.

The following steps help understand the issue:
1. Attacker generates fresh keys (A, B, C) for inception.
2. Attacker calls `registerKeyPairWithTransfer` with:
- `ctx.token = address(0)` (native token)
- `msg.value = 0`
- One native permit: `{token: address(0), amount: BRIDGE_BALANCE, recipients: []}`
3. Processing in `_executePermits`:
- `hasNativeTransfer = true` (native permit exists)
- Recipient loop: 0 recipients, `totalTransferred = 0`
- `transferOnly = true` (no wrap/unwrap flags)
- `remainder = BRIDGE_BALANCE - 0 = BRIDGE_BALANCE`
- `_transferNative(prerotatedKeyHash, BRIDGE_BALANCE)` — **sends entire bridge native tokens balance to attacker's next key**
- `totalTransferred = BRIDGE_BALANCE == permit.amount` — check passes
4. Key inception succeeds. Attacker receives all native tokens balance.

Impact

Complete drain of all native tokens held by the bridge

All users with wrapped native tokens become insolvent — cannot unwrap

Zero-cost attack (gas only) requiring only a fresh key inception

Recommendations

Add a cumulative `msg.value` check for native token operations:

```
uint256 nativeTokenUsed= 0;
nativeTokenUsed += recipient.amount;
// ... in the transfer loop for native tokens (line 324):
require(msg.value >= nativeTokenUsed, "Insufficient
native tokens");
// ... in the remainder path (line 335):
require(msg.value >= nativeTokenUsed, "Insufficient
native tokens");
```

YadaCoin

Confirmed.

Zealynx

Fixed.

7.2 High Severity Findings

[H-01] Transaction-level msg.value reused across multiple native wrap permits enables minting unbacked wrapped tokens

Affected files

`Bridge.sol` lines 358-393 (`_handleWrap`), 243-343 (`_executePermits`)

Description

When wrapping native token, the check `require(msg.value >= recipient.amount)` at line 370 is evaluated **per recipient and per permit**. Since `msg.value` is a transaction-level constant that never decreases, the same `msg.value` can pass the check for multiple wrap operations, minting more wrapped tokens than native tokens received.

Root Cause

```
// Bridge.sol _handleWrap line 370
require(msg.value >= recipient.amount, "Insufficient
native token sent");
```

This check passes independently for each wrap recipient/permit. Additionally, `totalTransferred` is reset to 0 at the start of each permit (line 303), so cross-permit accounting is absent.

Vulnerable Scenario:
The following steps help understand the issue:
1. Attacker sends `msg.value = 100` with **two native permits**, each containing one wrap recipient for 100 native tokens.
2. Permit 1: `require(msg.value(100) >= 100)` passes. Bridge mints ~100 wrapped tokens. `totalTransferred = 100 == permit.amount`. Check passes.

3. Permit 2: `require(msg.value(100) >= 100)` passes again. Bridge mints another ~100 wrapped tokens. `totalTransferred = 100 == permit.amount`. Check passes.
4. Result: Bridge received 100 native tokens, minted ~200 wrapped tokens. Solvency broken by 100 native tokens.

Impact

Native token solvency invariant violation: `address(bridge).balance >= wrappedNativeToken.totalSupply()` no longer holds

Attacker can unwrap their excess wrapped tokens to steal other users' escrowed native tokens

Repeatable for arbitrary (N permits × msg.value)

Recommendations

Track cumulative native tokens usage across ALL permits in a single transaction. Replace per-recipient `msg.value` check with a running counter:

```
uint256 nativeTokenUsed = 0;
nativeTokenUsed += recipient.amount;

// In _handleWrap for native tokens:
require(msg.value >= nativeTokenUsed , "Insufficient
native tokens");
```

YadaCoin

Confirmed.

Zealynx

Fixed.

[H-02] Unvalidated user-provided wrapped token address in pair registration enables binding legitimate tokens to malicious contracts

Affected files

`Bridge.sol` — `registerKeyPairWithTransfer()` (lines 470–560)

Description

Any user can register a token pair with a malicious wrapped token via `registerKeyPairWithTransfer`, permanently binding a legitimate original token (e.g., USDC or other not registered token) to an attacker-controlled ERC20. This enables complete theft of all wrapped funds for that token and permanently blocks the correct pair from ever being registered.

Root Cause
Two missing safeguards in the token pair registration logic inside `registerKeyPairWithTransfer`:

> **No access control on token pair registration.** The `newTokenPairs` field in `RegisterKeyPairContext` is processed for any caller during any key inception or rotation. There is no `onlyOwner` check.

> **User-provided wrapped token address is accepted without validation.** When `pair.wrappedToken != address(0)` (line 494), the address is stored directly into `tokenPairs` without verifying it was deployed by the bridge's beacon or that it enforces `onlyBridge` access control on `mint`/`burn`.

Vulnerable Scenario:

Prerequisites:
- The target original token (e.g., USDC) has not yet been registered as a pair on the bridge.
- The attacker deploys a standard ERC20 with no access control on `mint()` and `burn()`.

The following steps help understand the issue:
1. **Register malicious pair.** The attacker generates a fresh wallet and calls `registerKeyPairWithTransfer` for key inception, including `newTokenPairs` that maps USDC to their malicious ERC20. No special privileges are required.

Pair is permanently locked. Any subsequent attempt to register USDC with the correct wrapped token reverts with `TokenPairExists`. There is no removal or update function.

Victim wraps real tokens. A legitimate user **wraps** USDC through the bridge during their own key rotation. The bridge transfers real USDC from the victim into itself and mints the malicious ERC20 (which has no `onlyBridge` restriction) to the victim's next key. The victim receives worthless tokens.

Attacker mints fake tokens. The attacker freely mints any amount of the malicious ERC20 to themselves, since `mint()` has no access control.

Attacker drains real tokens. The attacker generates another fresh wallet, does key inception with an **unwrap** permit for the malicious token. The bridge burns

the worthless malicious tokens and releases real USDC from its reserves to the attacker. The bridge is fully drained for the affected token.

Impact

Fund theft: Attacker can drain 100% of the bridge's reserves for any affected token.

Permanent denial of service: The legitimate token pair can never be registered without a contract upgrade.

Recommendations

Separate token pair registration from key rotation and enforce access control:

1. Extract token pair registration into an `onlyOwner` function:

```
function registerTokenPair(
    address originalToken,
    string memory tokenName,
    string memory tokenSymbol
) external onlyOwner {
    if (tokenPairs[originalToken].wrappedToken !=
address(0)) revert TokenPairExists();

    bytes memory initData = abi.encodeWithSelector(
        WrappedToken.initialize.selector,
        tokenName,
        tokenSymbol,
        address(this),
```

```
            address(keyLogRegistry)
        );
        WrappedTokenProxy proxy = new
    WrappedTokenProxy(wrappedTokenBeacon, initData);
        address wrappedToken = address(proxy);

        tokenPairs[originalToken] =
    TokenPairData(originalToken, wrappedToken);
        tokenPairs[wrappedToken] =
    TokenPairData(originalToken, wrappedToken);
        supportedOriginalTokens.push(originalToken);
    }
```

2. Remove `newTokenPairs` from `RegisterKeyPairContext` and `registerKeyPairWithTransfer`.

3. Always deploy wrapped tokens via the beacon. Never accept a user-provided `wrappedToken` address. This guarantees every wrapped token enforces `onlyBridge` on `mint`/`burn`/`transfer`.

Additional Recommendations

> Add a `removeTokenPair` or `updateTokenPair` function so the owner can correct mistakes or respond to attacks.
>
> Remove the `wrappedToken` field from the `TokenPair` struct passed by users, since the bridge should always deploy it.

YadaCoin

Confirmed.

Zealynx

Fixed.

7.3 Medium Severity Findings

[M-01] ERC-2612 permit frontrunning allows griefing DoS on all permit-dependent Bridge operations

Affected files

Bridge.sol#L285-L293

Description

The `_executePermits` function calls `IERC20Permit2.permit()` directly without a try/catch wrapper:

```
IERC20Permit2(permit.token).permit(
    ectx.user,
    address(this),
    permit.amount,
    permit.deadline,
    permit.v,
    permit.r,
    permit.s
);
```

ERC-2612 permit signatures are visible in the mempool before inclusion. An attacker can extract the permit parameters from a pending transaction's calldata and frontrun by calling `permit()` directly on the ERC-20 token contract. The attacker's transaction succeeds (setting the allowance and consuming the nonce), so when the victim's Bridge transaction is included, the `permit()` call reverts because the signature/nonce is already used — causing the entire operation to fail.

This affects three critical Bridge functions that rely on `_executePermits`:

`registerKeyPairWithTransfer` — key pair registration with token transfers

`transferBalanceToLatestKey` — balance migration to latest rotated key

`upgradeWithKeyRotation` — contract upgrade with key rotation

The attacker cannot steal funds or profit from this attack, but can repeatedly grief any user attempting these operations, effectively blocking key registration, balance transfers, and upgrades that involve ERC-20 permits.

Impact

An attacker can prevent any permit-dependent Bridge operation from completing by frontrunning the permit signature. Users are unable to register key pairs with token transfers, migrate balances, or perform upgrades with key rotation until they work around the griefing by using a standard `approve` transaction instead — which requires an additional on-chain transaction and awareness of the issue.

Recommendations

Wrap the `permit()` call in a try/catch block. If the permit fails, check whether the allowance is already sufficient (which it will be if the attacker already submitted the permit). This is the standard mitigation used across the industry:

```
try IERC20Permit2(permit.token).permit(
    ectx.user,
    address(this),
    permit.amount,
    permit.deadline,
    permit.v,
    permit.r,
```

```
    permit.s
) {} catch {
    if (
        IERC20(permit.token).allowance(
            ectx.user,
            address(this)
        ) < permit.amount
    ) {
        revert InsufficientAllowance();
    }
}
```

YadaCoin

Confirmed.

Zealynx

Fixed.

[M-02] `rotateToPublicKey` only transfers Bridge ownership leads to permanent KeyLogRegistry desynchronization and retained privileged access by old owner

Affected files

Bridge.sol#L743-L758
KeyLogRegistry.sol#L60
KeyLogRegistry.sol#L168-L170
KeyLogRegistry.sol#L369

Description

Both Bridge and KeyLogRegistry independently track and transfer ownership during key rotation. In the normal flow via `registerKeyPairWithTransfer`, both contracts transfer ownership atomically in the same transaction: KeyLogRegistry does so internally in `registerKeyLogPair`, and Bridge does so explicitly afterward. The same applies to `upgradeWithKeyRotation`.

However, `rotateToPublicKey` only transfers Bridge ownership and does not trigger any ownership update on KeyLogRegistry:

```
function rotateToPublicKey(
    bytes memory existingOwnerPublicKey
) external {
    // ...
    feeCollector = latest.prerotatedKeyHash;
    transferOwnership(latest.prerotatedKeyHash);
}
```

After this call, the two contracts are permanently desynchronized:
- `Bridge.owner()` = new rotated address
- `KeyLogRegistry.owner()` = old address

Vulnerable Scenario:
The following steps help understand the issue:
1. The owner calls `rotateToPublicKey()` to rotate to the next key in the pre-rotation chain.
2. Bridge ownership transfers to the new pre-committed address.
3. KeyLogRegistry ownership remains at the old address.
4. The old address retains full control over KeyLogRegistry, including `setAuthorizedCaller()` and `_authorizeUpgrade()`.
5. If the old key is later compromised, the attacker can call `setAuthorizedCaller(attacker)` on KeyLogRegistry, revoking the Bridge's authorization and permanently bricking all key registration operations (`registerKeyLog` and `registerKeyLogPair` are `onlyAuthorized`).
6. The attacker can also upgrade KeyLogRegistry to a malicious implementation via `_authorizeUpgrade`, since they are still its owner.

This is especially concerning given the protocol's core purpose is protection against key theft. The pre-rotation scheme guarantees that a compromised key cannot steal funds, but `rotateToPublicKey` breaks this guarantee for KeyLogRegistry governance.

Impact

After `rotateToPublicKey`, the old owner retains full administrative control over KeyLogRegistry, including the ability to revoke Bridge authorization (bricking all operations) and upgrade KeyLogRegistry to a malicious implementation. This undermines the pre-rotation security model that the protocol is built upon.

Recommendations

Remove `rotateToPublicKey` entirely and force all ownership changes through `registerKeyPairWithTransfer`, which keeps both contracts in sync atomically. This function is the only ownership rotation path that causes desynchronization, and it is also the source of other issues (unchecked `exists` return value, forced fee collector overwrite). If the function must be preserved, add a corresponding KeyLogRegistry ownership transfer:

```
function rotateToPublicKey(
    bytes memory existingOwnerPublicKey
) external {
    // ... existing checks ...
    (KeyLogEntry memory latest, bool exists) =
        keyLogRegistry.getLatestChainEntry(
            existingOwnerPublicKey
        );
    require(exists, "Key log not initialized");
    feeCollector = latest.prerotatedKeyHash;
    transferOwnership(latest.prerotatedKeyHash);
    // Sync KeyLogRegistry ownership
    keyLogRegistry.transferOwnership(
        latest.prerotatedKeyHash
    );
}
```

YadaCoin

Confirmed. Added `KeyLogRegistry` ownership synchronization in `rotateToPublicKey` and introduced a `transferOwnershipForKeyRotation` wrapper to handle atomic ownership transfer across both contracts.

Zealynx

Fixed. Verified that ownership is now transferred atomically to both Bridge and KeyLogRegistry, preventing the desynchronization vector.

[M-03] Forced feeCollector update during ownership transfer leads to potential fee misdirection

Affected files

Bridge.sol#L203
Bridge.sol#L754

Description

Both `transferOwnership()` and `rotateToPublicKey()` automatically overwrite `feeCollector` with the new owner address as a side effect of transferring ownership. There is no way to transfer ownership without also redirecting all protocol fees.

Vulnerable Scenario:
The following steps help understand the issue:
1. The owner has `feeCollector` set to a dedicated fee-receiving address (e.g., a multisig or treasury).
2. The owner calls `rotateToPublicKey()` to rotate to the next key in the pre-rotation chain.
3. `feeCollector` is silently overwritten to `latest.prerotatedKeyHash` at line 754, before `transferOwnership()` overwrites it again at line 203.
4. All protocol fees now flow to the new owner address instead of the dedicated treasury.
5. The new owner must immediately call `setFeeCollector()` to restore the intended fee destination, creating a race window where fees can be misdirected.

Impact

Protocol fees are silently redirected to the new owner address on every ownership transfer or key rotation. If the new owner is not monitoring for this, fees accumulate in an unintended address. During the window between ownership transfer and a corrective `setFeeCollector()` call, any wrap/unwrap operations will send fees to the wrong destination.

Recommendations

Decouple the `feeCollector` update from ownership transfer. Remove `feeCollector = newOwner` from `transferOwnership()` and `feeCollector = latest.prerotatedKeyHash` from `rotateToPublicKey()`. If automatic fee redirection is desired, make it opt-in via a separate parameter or a dedicated function call after the transfer.

YadaCoin

Confirmed. Removed the `feeCollector` state variable entirely. Fees now flow directly to `owner()`, eliminating the forced overwrite issue.

Zealynx

Fixed. Confirmed the removal of `feeCollector` resolves the fee misdirection vector. Fresh deployment strategy addresses any storage layout concerns from the refactor.

[M-04] Gas DoS via Key Chain Traversal

Affected files

`KeyLogRegistry.sol` lines 252-312, 335-342

Description

`getLatestChainEntry` calls `buildFromPublicKey` → `_buildChainFromHash`, which performs two O(N) traversals every time it is called:
1. **Backward traversal** (lines 261-282): follows `prevPublicKeyHash` links to find inception
2. **Forward traversal** (lines 289-303): follows `prerotatedKeyHash` links from inception to the latest entry

Additionally, `_buildChainFromHash` allocates a fixed `KeyLogEntry[](1000)` array in memory on every call (line 253), even though the Bridge only ever needs the **latest** entry (`log[log.length - 1]`). The full array is never consumed by any on-chain caller — Bridge.sol calls only `getLatestChainEntry`

Measured cost: a single `getLatestChainEntry` call costs between 700k~900k gas at chain length 101 when called from the last key (Key chain length: each pair rotation adds 2 entries, so 50 rotations = chain length 101 including inception). The cost depends on the starting key's position in the chain. The Bridge passes `ctx.unconfirmed.publicKey` — a key near the end — requiring near-full backward traversal before the forward traversal.

Vulnerable Scenario:
The following steps help understand the issue:
1. A user registers key inception via `registerKeyLog` — chain length is now 1.
2. The user performs 50 key pair rotations via `registerKeyLogPair` — each rotation adds 2 entries (unconfirmed + confirming), growing the chain to 101 entries.
3. The user calls any bridge operation (e.g., `registerKeyPairWithTransfer`

to wrap tokens). Internally, the Bridge calls `getLatestChainEntry(ctx.unconfirmed.publicKey)` (line 512).
4. `_buildChainFromHash` receives the unconfirmed key (near the end of the chain) and:
- Traverses **backward** through ~100 entries to find inception (~100 SLOADs)
- Allocates `KeyLogEntry[](1000)` in memory
- Traverses **forward** through all 100 entries to find the latest (~100 SLOADs)
- Allocates a result array and copies all entries
5. Total cost: **~900k gas** for a single lookup — just to retrieve the latest entry.
6. With continued rotations, gas cost grows by ~4,400 per additional entry, eventually making bridge operations impractical or exceeding the block gas limit.

Impact

> Every bridge operation (wrap, unwrap, transfer, key rotation) pays O(N) gas for key chain lookup
>
> Users with long key chains eventually cannot perform any operations
>
> If the **owner's** chain grows too long, the bridge becomes unmanageable
>
> At ~1000 entries, chains hit the hardcoded array limit and silently truncate

Recommendations

Maintain two index mappings, updated during `registerKeyLog` / `registerKeyLogPair`, to make all lookups O(1):

```
mapping(address => address) public chainOf;        // any key hash → inception hash
```

```
mapping(address => address) public latestInChain;  // inception hash → latest key hash
```

This eliminates both traversals: given any mid-chain key, `chainOf` resolves to inception in O(1), and `latestInChain` resolves to the latest entry in O(1). The fixed 1000-element array allocation also becomes unnecessary.

YadaCoin

Confirmed.

Zealynx

Fixed.

7.4 Low Severity Findings

[L-01] No refund of excess `msg.value` in native BNB operations leads to permanent absorption of overpaid funds

Affected files

Bridge.sol#L369-L371
Bridge.sol#L458-L463

Description

In `_handleWrap` for native BNB operations, the balance check uses >= rather than ==:

```
require(
    msg.value >= recipient.amount,
    "Insufficient native token sent"
);
```

If `msg.value` exceeds the total BNB needed for the operation, the excess remains in the bridge's balance permanently. There is no refund mechanism anywhere in the transaction flow. The `totalTransferred != permit.amount` check at line 342 validates permit accounting but does not enforce that `msg.value` matches the required amount.

The only way to extract BNB from the bridge is `emergencyWithdrawBNB`, which sweeps the **entire** native balance — including collateral backing wrapped native tokens. It cannot selectively refund excess BNB without also removing collateral.

Recommendations

Either enforce an exact match or refund excess BNB at the end of the transaction:

```
// Option 1: Enforce exact amount
require(
    msg.value == recipient.amount,
    "Incorrect native token amount"
);

// Option 2: Refund excess after operations
uint256 excess = msg.value - totalRequired;
if (excess > 0) {
    _transferNative(msg.sender, excess);
}
```

YadaCoin

Confirmed.

Zealynx

Fixed.

[L-02] Absence of emergency pause mechanism across all contracts leads to inability to halt operations during active exploits

Affected files

Bridge.sol#L1-L15
KeyLogRegistry.sol#L1-L6
WrappedToken.sol#L1-L8

Description

Neither Bridge, KeyLogRegistry, nor any token contract inherits from `PausableUpgradeable`. There is no circuit breaker anywhere in the system. This is particularly significant given the protocol's specific risk profile.

During an active exploit or discovered vulnerability, the protocol has no way to immediately halt operations. The only available response is a full contract upgrade, which requires time to prepare and an uncompromised owner key — neither of which may be available during an emergency.

Recommendations

Add `PausableUpgradeable` to both Bridge and KeyLogRegistry with `whenNotPaused` on critical state-changing functions (`registerKeyPairWithTransfer`, `registerKeyLog`, `registerKeyLogPair`, `transferBalanceToLatestKey`, `upgradeWithKeyRotation`, `unwrap`). Consider introducing a separate `guardian` role (e.g., a multisig) with the ability to pause but not upgrade, providing defense-in-depth if the owner key is compromised.

YadaCoin

Confirmed. Implemented an upgrade-based approach with bridge-only upgrade gates and atomic multi-contract upgrades via `upgradeWithKeyRotation`,

providing emergency response capability.

Zealynx

Fixed. Verified the upgrade refactor after identifying and resolving revert issues in the initial implementation where all four upgrade paths were reverting.

[L-03] Hardcoded 18-Decimal Wrapped Tokens Break Unwrap Fees and Wallet Displays

Affected files

WrappedToken.sol#L17-L27
Bridge.sol#L410
Bridge.sol#L373

Description

`WrappedToken.initialize` never accepts or sets custom decimals. `ERC20Upgradeable` defaults to 18 decimals for all wrapped tokens regardless of the original token's decimals.

The Bridge mints and burns raw amounts directly without decimal conversion. This creates two concrete problems:

> **Asymmetric fee calculation between wrap and unwrap.** In `_handleWrap`, the fee is computed using `permit.token` (the original token), so `decimals` is correct. In `_handleUnwrap`, the fee is computed using `permit.token` (the wrapped token), which is always 18:

```
// _handleUnwrap — permit.token is the wrapped token
uint8 decimals = (permit.token == address(0))
    ? 18
    : IERC20WithDecimals(permit.token).decimals();
uint256 maxFeeRate = 10 ** decimals;
uint256 tokenFee =
    (recipient.amount * uctx.feeInfo.fee) / maxFeeRate;
```

For a 6-decimal original token (e.g., USDT on some chains): wrapping uses `maxFeeRate = 10^6` (correct), but unwrapping uses `maxFeeRate = 10^18` (fees become near-zero). The protocol loses fee revenue on every unwrap of non-18-decimal token pairs.

> **Misleading wallet and explorer display.** Wrapping 1 USDC (1,000,000 raw units at 6 decimals) mints 1,000,000 raw wrapped tokens interpreted as 18 decimals, displaying as 0.000000000001 in wallets and block explorers.

While most BSC tokens use 18 decimals, the protocol has no validation preventing registration of non-18-decimal tokens.

Recommendations

Pass the original token's decimals to `WrappedToken.initialize` and override `decimals()` to match:

```
uint8 private _decimals;

function initialize(
    string memory name,
    string memory symbol,
    address _bridge,
    uint8 decimals_
) public initializer {
    __ERC20_init(name, symbol);
    __ERC20Permit_init(name);
    __Ownable_init(_bridge);
    __UUPSUpgradeable_init();
    bridge = _bridge;
    _decimals = decimals_;
}
```

```
function decimals()
    public view override returns (uint8)
{
    return _decimals;
}
```

YadaCoin

Confirmed. Added a `decimals` parameter to `WrappedToken.initialize` and stored it in a `_decimals` state variable with a corresponding `decimals()` override.

Zealynx

Fixed. Verified the fix including alignment of the `Factory.createToken` parameters to pass the decimals value through the full token creation flow.

[L-04] No mechanism to deregister or disable token pairs leads to permanent persistence of compromised or deprecated pairs

Affected files

Bridge.sol#L490-L508
Bridge.sol#L492

Description

Once a token pair is registered via `registerKeyPairWithTransfer`, it is permanent. There is no function in the Bridge to remove a pair from `tokenPairs`, remove an entry from `supportedOriginalTokens`, disable wrap/unwrap for a specific pair, or blacklist a compromised wrapped token.

Additionally, the `TokenPairExists` check at line 492 prevents re-registration, meaning the owner cannot point an original token to a new wrapped token even if the existing one is compromised:

```
if (
    tokenPairs[pair.originalToken].wrappedToken
        != address(0)
) {
    revert TokenPairExists();
}
```

This becomes problematic if:
- A wrapped token's beacon implementation is compromised and all wrapping/unwrapping through that pair should be halted immediately.
- An original token is deprecated, migrated, or becomes unsafe (e.g., exploited on BSC), yet the Bridge continues supporting operations through the stale pair.
- A malicious or incorrect pair is registered by mistake. The only recourse is a full contract upgrade.

Recommendations

Add pair lifecycle management:

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

function disableTokenPair(
    address token_
) external onlyOwner {
    if (tokenPairs[token_].wrappedToken == address(0))
        revert TokenPairNotSupported();
    disabledPairs[token_] = true;
}

function enableTokenPair(
    address token_
) external onlyOwner {
    disabledPairs[token_] = false;
}
```

Then add a check at the start of `_handleWrap` and `_handleUnwrap`:

```
require(
    !disabledPairs[permit.token],
    "Token pair disabled"
);
```

YadaCoin

Confirmed. Implemented `deregisterTokenPair` with swap-and-pop removal from the `tokenPairs` array, fully removing deprecated or compromised pairs.

Zealynx

Fixed. Verified the deregistration mechanism replaces the initial disable/enable approach, addressing all identified gaps including unwrap bypass, one-directional coverage, and re-registration handling.

[L-05] Silent chain truncation at 1000 entries causes `getLatestChainEntry` to return stale data with no error indication

Affected files

KeyLogRegistry.sol#L252-L311
KeyLogRegistry.sol#L300-L302
KeyLogRegistry.sol#L335-L342

Description

`_buildChainFromHash` allocates a fixed-size array of 1000 elements and silently breaks the forward traversal when the index reaches the limit:

```
KeyLogEntry[] memory log = new KeyLogEntry[](1000);
// ...
if (logIndex >= log.length) {
    break;
}
```

If a key chain exceeds 1000 entries (~500 pair rotations), the function returns a truncated array. `getLatestChainEntry` then returns `log[999]` — **not the actual latest entry** — with no revert and no indication that the data is stale.

Every downstream consumer silently operates on incorrect data:

- **`isValidOwnershipTransfer`** (line 364) validates against the wrong entry, potentially accepting or rejecting ownership transfers incorrectly.
- **`rotateToPublicKey`** in Bridge reads the wrong `prerotatedKeyHash`, transferring Bridge ownership to the wrong address.

`transferBalanceToLatestKey` in Bridge sends all token balances to an outdated key.

`registerKeyPairWithTransfer` in Bridge routes permits to the wrong `prerotatedKeyHash`.

The failure is completely silent — no revert, no event, no return flag. The `bool` return from `getLatestChainEntry` is `true` because the array is non-empty, even though the entry is stale.

While 500 pair rotations is unlikely short-term, the protocol provides no guard against reaching this boundary over its deployment lifetime, and the consequences are catastrophic if it does.

Recommendations

Either revert if the chain exceeds the limit:

```
if (logIndex >= log.length) {
    revert("Chain exceeds maximum length");
}
```

YadaCoin

Confirmed.

Zealynx

Fixed.

[L-06] Missing `_disableInitializers()` in constructor allows initialization of implementation contracts

Affected files

WrappedToken.sol
MockERC20.sol
Bridge.sol
WrappedTokenFactory.sol

Description

`KeyLogRegistry` correctly prevents initialization of the implementation contract by calling `_disableInitializers()` in its constructor:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

However, four other UUPS upgradeable contracts — `Bridge`, `WrappedToken`, `MockERC20`, and `WrappedTokenFactory` — have no constructor and therefore leave their implementation contracts initializable.

This means anyone can call `initialize()` directly on the bare implementation contract (not the proxy), setting arbitrary values for `bridge`, `owner`, and other initialization parameters. While this does not affect existing proxies (each proxy has its own storage), it violates the defense-in-depth pattern that OpenZeppelin explicitly recommends for all UUPS contracts.

Recommendations

Add the standard constructor to all UUPS upgradeable contracts:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

YadaCoin

Confirmed.

Zealynx

Fixed.

[L-07] Early loop break in `_executePermits` causes order-dependent revert on valid owner transactions

Affected files

Bridge.sol#L222-L241

Description

The `_executePermits` function scans the `permits[]` array in a single pass to detect two independent conditions:

- Whether a native transfer permit exists (`hasNativeTransfer`)
- Whether any recipient requires mint/burn privileges (`requiresOwner`)

However, when condition (2) is found, the loop immediately `breaks`:

```
for (uint256 i = 0; i < ectx.permits.length; i++) {
    PermitData memory permit = ectx.permits[i];
    if (permit.token == address(0)) {
        hasNativeTransfer = true;
    }
    if (permit.token == ectx.token) {
        for (uint256 j = 0; j < permit.recipients.length;
j++) {
            Recipient memory recipient =
permit.recipients[j];
            if (recipient.mint || recipient.burn) {
                requiresOwner = true;
                break;
            }
```

```
            }
            if (requiresOwner) break; // exits outer loop
        }
    }
    if (!hasNativeTransfer) revert MissingPermit();
```

Valid owner transactions (direct mint/burn) revert with a misleading `MissingPermit()` error if the permits array is not ordered with the native transfer entry before the ERC-20 mint/burn entry. While not exploitable by attackers, it creates a fragile integration surface and confusing debugging experience.

Recommendations

Separate the two concerns by removing the early break, allowing the full array to be scanned:

```
for (uint256 i = 0; i < ectx.permits.length; i++) {
    PermitData memory permit = ectx.permits[i];
    if (permit.token == address(0)) {
        hasNativeTransfer = true;
    }
    if (!requiresOwner && permit.token == ectx.token) {
        for (uint256 j = 0; j < permit.recipients.length;
j++) {
            if (permit.recipients[j].mint
                || permit.recipients[j].burn) {
                requiresOwner = true;
                break;
            }
        }
    }
}
```

This preserves the inner `break` optimization while ensuring the full permits array is always traversed for the native transfer check.

YadaCoin

Confirmed.

Zealynx

Fixed.

[L-08] `_handleMint` Fee Is Computed but Never Collected

Affected files

Bridge.sol#346-356

Description

`_handleMint` calculates a `tokenFee` and subtracts it from the minted amount, but never transfers the fee to `feeCollector` or anyone else

Recommendations

Mint the fee to `feeCollector`:

```
function _handleMint(...) internal {
    ...
    uint256 tokenFee = (recipient.amount *
mctx.feeInfo.fee) / maxFeeRate;
    uint256 netAmount = recipient.amount - tokenFee;

IMockERC20(permit.token).mint(recipient.recipientAddress,
netAmount);
    if (tokenFee > 0)
IMockERC20(permit.token).mint(feeCollector, tokenFee);
}
```

YadaCoin

Confirmed.

Zealynx

Fixed.

[L-09] Mandatory Native Permit Check Bypassable with Zero Value

Affected files

`Bridge.sol` lines 224-240

Description

The pre-flight validation loop in `_executePermits` requires a native BNB permit (`token == address(0)`) to exist in the permits array:

```
// Bridge.sol lines 222-240
for (uint256 i = 0; i < ectx.permits.length; i++) {
    PermitData memory permit = ectx.permits[i];
    if (permit.token == address(0)) {
        hasNativeTransfer = true;
    }
    // ...
}
if (!hasNativeTransfer) revert MissingPermit();
```

However, this check only validates the **structural presence** of a native permit in the array — it does not enforce:
- `msg.value > 0`
- `permit.amount > 0`
- That the permit has any recipients

Since `registerKeyPairWithTransfer` is `payable`, it accepts `msg.value = 0`. A caller can satisfy the mandatory native permit check with a no-op entry:

```
{ token: address(0), amount: 0, deadline: 0, v: 0, r: 0,
s: 0, recipients: [] }
```

The strict accounting check (`totalTransferred != permit.amount`) passes because `0 == 0`.

If the intent was to force every transaction to include a BNB payment, this does not achieve it — users can bypass it with a zero-amount native permit

Recommendations

If BNB payment is mandatory, add an explicit `msg.value` floor:

```
if (!hasNativeTransfer) revert MissingPermit();
require(msg.value > 0, "Native BNB payment required");
```

YadaCoin

Confirmed.

Zealynx

Fixed.

[L-10] Broken Logic in `Bridge::getLatestEntryByPrerotatedKeyHash`

Affected files

KeyLogRegistry.sol#L236-L250

Description

The function `Bridge::getLatestEntryByPrerotatedKeyHash` is intended to return the latest key rotation entry for a given `prerotatedKeyHash`. However, the logic is backwards. It checks:

```
require(idx == 0, "Not the latest key rotation.");
```

Here, `idx` comes from `byPublicKeyHash[currentAddress]`. This should be the latest key as `idx` is supposed to be non zero, and so the function fails and reverts.

Recommendation

Remove the `require` statement and simply return the entry:
```solidity
function getLatestEntryByPrerotatedKeyHash(address prerotatedKeyHash) public view returns (KeyLogEntry memory, bool) {
uint256 idx = byPrerotatedKeyHash[prerotatedKeyHash];
if (idx == 0) {
idx = byTwicePrerotatedKeyHash[prerotatedKeyHash];
if (idx == 0) {
KeyLogEntry memory emptyEntry;
return (emptyEntry, false);
}
```

```
}
KeyLogEntry memory entry = keyLogEntries[idx - 1];
return (entry, true);
}
```

YadaCoin

Confirmed.

Zealynx

Fixed.

[L-11] Owner Can Drain Bridge via emergencyWithdrawBNB

Affected files

`Bridge.sol` lines 458-463

Description

Centralization Risk. The owner can unilaterally withdraw **all** native BNB from the bridge via `emergencyWithdrawBNB`, including BNB escrowed to back wrapped tokens. No solvency check, timelock, or multisig exists.

Recommendations

Consider adding a timelock, e.g. 48-hour delay publicly visible on-chain, giving users time to unwrap and exit if they disagree with the withdrawal. Alternatively, add a solvency check (`address(this).balance – wrappedBNBToken.totalSupply()`) to limit withdrawals to excess BNB only.

YadaCoin

Confirmed.

Zealynx

Fixed.

[L-12] feeCollector Silently Overwritten by Arbitrary User

Affected files

`Bridge.sol` lines 744-758 (`rotateToPublicKey`)

Description

`rotateToPublicKey` (line 755): Sets `feeCollector = latest.prerotatedKeyHash` before calling `transferOwnership`. This function is permissionless — anyone can call it with the owner's public key (readable from on-chain events). However, the rotation always goes to the owner's pre-committed `prerotatedKeyHash`, so no unauthorized party gains control.
If the owner uses `setFeeCollector` to route fees to a separate cold wallet or multisig, any subsequent key rotation (or an arbitrary user calling `rotateToPublicKey`) resets `feeCollector` to the new owner address.

Recommendations

Restrict `rotateToPublicKey` with `onlyOwner` modifier

Or make `feeCollector` to be controlled only by `setFeeCollector`

Also adding `require(exists, "Key log not initialized")` would make an explicit revert rather than depending on downstream revert in `transferOwnership(latest.prerotatedKeyHash);`

YadaCoin

Confirmed.

Zealynx

Fixed.

[L-13] Fixed 30,000 gas stipend in `_transferNative` leads to potential incompatibility with contract recipients

Affected files

Bridge.sol#L162
Bridge.sol#L453-L456
Bridge.sol#L461

Description

All native BNB transfers in the protocol go through `_transferNative`, which uses a hardcoded gas stipend of 30,000:

```
uint256 private constant GAS_LIMIT = 30000;

function _transferNative(
    address to, uint256 amount
) private {
    (bool success, ) =
        to.call{value: amount, gas: GAS_LIMIT}("");
    if (!success) revert TransferFailed();
}
```

While 30,000 gas is sufficient for EOAs and most common contract wallets (including Gnosis Safe), certain smart contract wallets or custom contracts with heavier `receive()`/`fallback()` logic could exceed this stipend. If `feeCollector` or a user's `prerotatedKeyHash` resolves to such a contract, all native BNB operations involving that address would revert.

This affects fee collection in `_handleWrap` (line 377), remainder transfers in `_handleWrap` (line 387), unwrap payouts in `_handleUnwrap` (lines 416-417), and plain native transfers (line 324).

Notably, `emergencyWithdrawBNB` at line 461 uses **no gas limit** at all:

```
(bool sent, ) = to.call{value: balance}("");
```

This inconsistency suggests the gas limit may not be intentional across the board. The reentrancy risk that `GAS_LIMIT` was likely intended to mitigate is already handled by the `nonReentrant` modifier on all entry points.

Impact

Native BNB operations (wrap, unwrap, transfer) could fail if the recipient is a contract whose `receive()/fallback()` exceeds 30,000 gas. This would block fee collection or balance migration for affected addresses.

Recommendations

Since all external entry points already use `nonReentrant`, consider removing the gas stipend or increasing it significantly (e.g., 100,000) to accommodate a wider range of contract recipients. Alternatively, align `_transferNative` with the pattern used in `emergencyWithdrawBNB` by removing the gas limit entirely.

YadaCoin

Confirmed.

Zealynx

Fixed.

[L-14] Missing enforcement of `Bridge::MAX_TOKEN_PAIRS`

Affected files

Bridge.sol#L508

Description

The Bridge.sol contract defines a `MAX_TOKEN_PAIRS` constant but does not enforce it when new token pairs are registered. In `Bridge::registerKeyPairWithTransfer` the code pushes tokens into the `Bridge::supportedOriginalTokens` array without checking length limits or duplicates. This allows unbounded growth of the array.

Recommendations

Enforce the `MAX_TOKEN_PAIRS` limit when registering new pairs:

```
require(supportedOriginalTokens.length < MAX_TOKEN_PAIRS,
"max token pairs reached");
```

YadaCoin

Confirmed.

Zealynx

Fixed.

[L-15] Missing fee-on-transfer accounting in `_handleWrap` leads to potential under-collateralization of wrapped tokens

Affected files

Bridge.sol#L379-L382

Description

In `_handleWrap`, the bridge mints wrapped tokens based on the **nominal** transfer amount without verifying how many original tokens were actually received:

```
IERC20(pair.originalToken).safeTransferFrom(
    wctx.user, address(this), recipient.amount - tokenFee
);
if (tokenFee > 0) {
    IERC20(pair.originalToken).safeTransferFrom(
        wctx.user, feeCollector, tokenFee
    );
}
WrappedToken(pair.wrappedToken).mint(
    wctx.prerotatedKeyHash, recipient.amount - tokenFee
);
```

If the original token applies a transfer tax (fee-on-transfer), the bridge receives fewer tokens than `recipient.amount - tokenFee`, but still mints the full nominal amount of wrapped tokens. Over time, this creates a collateral deficit: more wrapped tokens exist than the bridge holds in original tokens.

Token pair registration via `registerKeyPairWithTransfer` has no token-type validation or whitelist, so nothing prevents a fee-on-transfer token from being registered. This is particularly relevant on BSC, where fee-on-transfer tokens (SafeMoon-style) are prevalent.

If a fee-on-transfer token is registered as an original token, the bridge gradually becomes under-collateralized for that pair. Late unwrappers would find insufficient original tokens in the bridge to redeem their wrapped tokens.

Recommendations

Use balance-before/after accounting to mint only the actually received amount:

```
uint256 balBefore = IERC20(pair.originalToken)
    .balanceOf(address(this));
IERC20(pair.originalToken).safeTransferFrom(
    wctx.user, address(this), recipient.amount - tokenFee
);
uint256 actualReceived = IERC20(pair.originalToken)
    .balanceOf(address(this)) - balBefore;
WrappedToken(pair.wrappedToken).mint(
    wctx.prerotatedKeyHash, actualReceived
);
```

YadaCoin

Confirmed.

Zealynx

Fixed.

[L-16] Extra Argument in `WrappedToken` Initialization is Ignored

Affected files

Bridge.sol#L496-L502
WrappedToken.sol#L17-L27
WrappedTokenFactory.sol#L12-L19
WrappedTokenFactory.sol#L39-L46

Description

In `Bridge::registerKeyPairWithTransfer` the code constructs `initData` for the new wrapped token proxy using `abi.encodeWithSelector` and passes four arguments (name, symbol, bridge address, keyLogRegistry address):

```
bytes memory initData = abi.encodeWithSelector(
    WrappedToken.initialize.selector,
    pair.tokenName,
    pair.tokenSymbol,
    address(this),
    address(keyLogRegistry)
);
WrappedTokenProxy proxy = new
WrappedTokenProxy(wrappedTokenBeacon, initData);
```

However, the `WrappedToken` implementation `initialize` signature expects only three parameters:

```
function initialize(
    string memory name,
    string memory symbol,
    address _bridge
) public initializer {
```

```
    __ERC20_init(name, symbol);
    __ERC20Permit_init(name);
    __Ownable_init(_bridge);
    __UUPSUpgradeable_init();
    bridge = _bridge;
}
```

This is also found in `WrappedTokenFactory::createToken` where the Interface `IWrappedToken` does not match the implementation (`WrappedToken::initialize`).

Recommendations

> Make the initialize function in `WrappedToken.sol` accept the extra `keyLogRegistry` parameter:

```
function initialize(
    string memory name,
    string memory symbol,
    address _bridge,
    address _keyLogRegistry
) public initializer {
    __ERC20_init(name, symbol);
    __ERC20Permit_init(name);
    __Ownable_init(_bridge);
    __UUPSUpgradeable_init();
    bridge = _bridge;
    keyLogRegistry = KeyLogRegistry(_keyLogRegistry);
}
```

YadaCoin

Confirmed.

Zealynx

Fixed.

7.5 Informational Severity Findings

[I-01] Duplicate Public Key Length Validation in `Bridge::registerKeyPairWithTransfer`

Description

In `Bridge::registerKeyPairWithTransfer`, the public key length is validated more than once. First, the contract calls:

```
address unconfirmedPublicKey =
getAddressFromPublicKey(ctx.unconfirmed.publicKey);
```

Inside `Bridge::getAddressFromPublicKey`, the length is already checked:

```
function getAddressFromPublicKey(bytes memory publicKey)
public pure returns (address) {
    if (publicKey.length != PUBLIC_KEY_LENGTH) revert
InvalidPublicKey();
```

Later in the same function, the code checks the length again:

```
if (ctx.unconfirmed.publicKey.length !=
PUBLIC_KEY_LENGTH)
    revert InvalidPublicKey();
```

Since `Bridge::getAddressFromPublicKey` already reverts if the length is incorrect, these additional length checks are unnecessary and will never catch anything new.

Recommendation

Remove the duplicate length checks and rely on `Bridge::getAddressFromPublicKey` to handle validation.

YadaCoin

Confirmed.

Zealynx

Fixed.

[I-02] Redundant Zero Address Check in `Bridge::registerKeyPairWithTransfer` and `Bridge::rotateToPublicKey`

Description

Two locations contain unreachable `ZeroAddress()` reverts that can never trigger due to prior guards.

Location 1 — `registerKeyPairWithTransfer`

Inside the confirming logic, the code first checks that the `outputAddress` is non-zero:

```
if (ctx.confirming.outputAddress != address(0)) {
```

Then inside that same block, it checks again if the address is zero:

```
if (ctx.confirming.outputAddress == address(0))
    revert ZeroAddress();
```

This second check can never be true. If execution entered the block, it already means `ctx.confirming.outputAddress != address(0)`. The inner revert is dead code.

Location 2 — `rotateToPublicKey`

The code checks whether the derived owner address is zero:

```
if (existingOwnerAddress == address(0)) revert
```

```
ZeroAddress();
```

However, this address is obtained from:

```
address existingOwnerAddress =
    getAddressFromPublicKey(ctx.unconfirmed.publicKey);
```

And `getAddressFromPublicKey` strictly enforces a 64-byte public key before hashing with `keccak256`. Getting `address(0)` would require the hash to end with 20 zero bytes — for a valid 64-byte input, the probability is ~1/2^160, which is practically impossible.

Recommendations

> Remove the redundant inner zero-address check in `registerKeyPairWithTransfer`.
>
> Remove the zero-address check in `rotateToPublicKey`.

YadaCoin

Confirmed.

Zealynx

Fixed.

[I-03] Dead Code in `_handleWrap` and `_handleUnwrap`

Description

`totalTransferred` is passed by value to `_handleWrap` and `_handleUnwrap`. The remainder code inside these functions updates a local copy that is never propagated back to `_executePermits`.

Recommendations

Remove the remainder code from _handleWrap and _handleUnwrap. The `transferOnly` remainder path at lines 331-340 already handles remainder forwarding correctly at the `_executePermits` level. If partial wraps with remainder forwarding are desired in the future, return `totalTransferred` from the handlers.

YadaCoin

Confirmed.

Zealynx

Fixed.