WrappedCBDC Stablecoin - cNGN (Solidity)

Audit Report

18th June 2025

FAILSAFE

Table of Contents

Executive Summary

This audit covers the Solidity implementation of the cNGN stablecoin ecosystem, including the core token (Cngn & Cngn2: upgradeable version), meta-transaction forwarder (Forwarder), permissioned roles (Admin/Operations), and multisignature wallet (MultiSig) modules.

Our review identified two areas of focus: the current implementation of emergency-pause controls on direct transfers, and the meta-transaction sender resolution approach that combines relayer and governance authorities.

Project Details

Project	WrappedCBDC Stablecoin - cNGN (Solidity)
URL	https://cngn.co/
Source Code	https://github.com/wrappedcbdc/stablecoin-cngn
Initial Commit	df82ba1d3a6837403fc649689a6b276adfb2bf2f
Interim Commit	cbf142b19b916504870f2c016f20f0bbd29cbfa7
Final Commit	5bcd4541d9a2952cf7edae47f47305d5d0a5c2eb
Timeline	13th May 2025 - 18th June 2025

Structure & Organization of Audit Report

Issues are tagged as “Open”, “Acknowledged”, “Partially Resolved”, “Resolved” or “Closed” depending on whether they have been fixed or addressed.

- Open: The issue has been reported and is awaiting remediation from developer team.
- Acknowledged: The developer team has reviewed and accepted the issue but has decided not to fix it.
- Partially Resolved: Mitigations have been applied, yet some risks or gaps still remain.
- Resolved: The issue has been fully addressed and no further work is necessary.
- Closed: The issue is deemed no longer pertinent or actionable.

Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical	The issue affects the Smart Contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
High	The issue affects the ability of the Smart Contract to compile or operate in a significant way.
Medium	The issue affects the ability of the Smart Contract to operate in a way that doesn’t significantly hinder its behavior.
Low	The issue has minimal impact on the Smart Contract’s ability to operate.
Info	The issue is informational in nature and does not pose any direct risk to the Smart Contract’s operation.

Audit Approach

The following are areas of concern will be investigated during the audit, along with any similar potential issues:

- Correctness of the implementation;
- Adversarial actions and other attacks on the network;
- Potential misuse and gaming of the Smart Contracts;
- Attacks that impacts funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution of the Smart Contracts;
- Vulnerabilities in the Smart Contracts code;
- Protection against malicious attacks and other ways to exploit Smart Contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

The following schedule will be followed:

- Code review completed and delivery of Initial Audit Report
- Client responds with fixes and/or acknowledgments for all findings
- Security team validates the fixes and/or acknowledgments
- Verification completed
- Delivery of Final Audit Report

Project Goals

1. Ensure Complete Pause Coverage: Verify that every token movement and state-altering function honors the global pause flag to fully halt operations during emergencies.
2. Standardize Meta-Transaction Context: Adopt a battle-tested ERC-2771 context implementation so that _msgSender() unambiguously and securely maps to the original user in all entrypoints.
3. Validate Upgradeable Patterns: Confirm that UUPS/initializer patterns correctly set and guard all critical state fields (e.g., trustedForwarderContract, adminOperationsContract) and enforce owner constraints.
4. Preserve Token Economic Invariants: Audit the external->internal "redemption" path to guarantee it neither burns tokens silently nor emits misleading transfer events, thereby maintaining total supply integrity.
5. Harden Access Control Separation: Enforce clear separation between gas-payer roles (forwarders/relayers) and privileged roles (minters/pausers/blacklisters) to prevent escalation of compromised keys.
6. Rigorous ABI & Bounds Checking: Ensure all on-chain decoding (EIP-712, custom calldata slices) includes explicit bounds, domain separation, and program-ID verifications to prevent instruction spoofing or panics.
7. Streamline Event Semantics: Align emitted events with actual state changes (e.g., distinguish burns vs. transfers) to provide an accurate on-chain audit trail for downstream consumers.

Audit Methodology

FailSafe employs a multi-layered approach to Smart Contract security audits:

Threat Modelling: We identify critical assets, enumerate potential threats, assess vulnerabilities, and prioritize risks based on severity and impact.

Manual Code Review: Our experts conduct a detailed, line-by-line review of the code, analyzing business logic, access controls, gas efficiency, and external dependencies.

Functional Testing: Using frameworks like Hardhat and Foundry, we perform comprehensive functional and integration tests to ensure correct and secure Smart Contract behavior.

Fuzzing & Invariant Testing: Advanced techniques such as fuzzing and invariant testing are used to uncover hidden vulnerabilities and verify Smart Contract consistency under diverse scenarios.

Edge Case Analysis: We rigorously test for extreme inputs, exception handling, concurrency, and non-standard scenarios to ensure robust Smart Contract performance.

Reporting & Recommendations: Our reports clearly describe each issue, its impact, location, root cause, and provide actionable remediation steps and best practice guidelines.

Remediation Support: We work closely with your team to implement and validate fixes, followed by a final assessment to confirm all issues are resolved.

FailSafe's process ensures your Smart Contracts are secure from initial deployment through ongoing operation, providing proactive and comprehensive protection.

In-scope Files

All Solidity contracts under contracts/, including:

- Cngn.sol, Cngn2.sol
- Forwarder.sol
- Operations.sol, Operations2.sol
- Multisig.sol

Out of Scope

- External dependencies (OpenZeppelin libraries).

- Off-chain infrastructure (relayer services, front-end code).
- Integration tests, deployment scripts, or proxy configurations beyond what is directly coded in the audited contracts.

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved	Closed
Critical	-	-	-	-	-	-
High	1	-	1	-	-	-
Medium	2	-	-	-	2	-
Low	2	-	-	-	2	-
Info	-	-	-	-	-	-
Total	5	0	1	0	4	0

#	Findings	Severity	Status
1	External-to-Internal Redemption Flow Results in Unintended Token Destruction	High	Acknowledged
2	Non-Sequential Nonce Validation in Meta-Transaction System	Medium	Resolved
3	Custom Meta-Transaction Implementation Requires Security Enhancement	Medium	Resolved
4	Incomplete Pause Mechanism Implementation	Low	Resolved
5	Inconsistent Meta-Transaction Context Implementation	Low	Resolved

Finding 1: External-to-Internal Redemption Flow Results in Unintended Token Destruction

Severity: High

Status: Acknowledged

Source: cngn.sol (transfer(address,uint256))

Description:

The transfer function's special-case branch for external-to-internal "redemption" flow currently results in token destruction without proper accounting. Under this path, tokens are transferred from an externally whitelisted sender to an internally whitelisted recipient and are subsequently burned from the recipient's account. The implementation currently credits the recipient's balance and emits a standard Transfer(from, to, amount) event. Off-chain monitoring tools will observe this standard event and assume the recipient was credited, while on-chain the tokens are permanently removed from circulation.

The implementation currently does not include documentation or comments explaining the total supply preservation mechanism or the rationale for maintaining unchanged balances. This implementation differs from the ERC-20 invariant ("total supply equals sum of balances") and may not align with user expectations, potentially leading to confusion and unintended token loss.

Impact:

In production, users transferring funds to internal accounts may discover their balance has been reduced without the recipient receiving the tokens. Each such transaction reduces the total supply, creating a mechanism that could be utilized to manipulate token scarcity or affect liquidity. Downstream services (wallets, explorers, accounting systems) will report incorrect balances and supply, potentially impacting trust and exposing the protocol to regulatory and compliance considerations.

Code:

```
/**
 * @dev Transfers tokens to a specified address.
 *
 * Special case: If the recipient is an internal whitelisted user and the sender is
 * an external whitelisted sender, the tokens are transferred and then immediately burned.
 * This represents a redemption flow where external users can send tokens to internal users
 * who then redeem them (burn).
 *
```

```
 * @param to The address to transfer to
 * @param amount The amount to be transferred
 * @return bool Returns true for a successful transfer
 */
function transfer(
    address to,
    uint256 amount
) public virtual override nonReentrant returns (bool) {
    address owner = _msgSender();

    // Check for blacklisted addresses
    require(!IAdmin(adminOperationsContract).isBlackListed(owner), "Sender is
        blacklisted");
    require(!IAdmin(adminOperationsContract).isBlackListed(to), "Recipient is
        blacklisted");

    // Special case: Redemption flow
    if (
        IAdmin(adminOperationsContract).isInternalUserWhitelisted(to) &&
        IAdmin(adminOperationsContract).isExternalSenderWhitelisted(owner)
    ) {
        // Transfer to internal user and then burn (redemption)
        _transfer(owner, to, amount);
        _burn(to, amount);
    } else {
        // Standard transfer
        require(!IAdmin(adminOperationsContract).isBlackListed(_msgSender()));
        require(!IAdmin(adminOperationsContract).isBlackListed(to));
        _transfer(owner, to, amount);
    }

    return true;
}
```

Remediation:

Consider implementing a clear two-step redemption API with distinct event sequences. Adding comprehensive documentation of the flow, implementing separate permissions for "redeem" operations, and ensuring total supply and balances remain fully traceable would enhance the system's reliability.

Developer Response:

The external-to-internal redemption flow is an intentional implementation for their bridge/cross-chain system. When internal whitelisted addresses receive funds, tokens are burned and equivalent amounts are minted on other blockchain networks through their chain indexer system. Developer have safeguard measures in place where users can only mint what's allowed and burn from their own wallet addresses. The centralization is necessary for their current architecture - their chain indexer tracks transactions and their API system handles burning and minting with a multi-sig warm wallet design.

Auditor Response:

The design relies on centralized chain indexers which carries industry-wide centralization risks. The audit team recommended on-chain and off-chain safeguards with clear documentation, however Developer ultimately acknowledged that the necessary components for the cNGN token transfer method require a centralized approach. The audit team marked the issue as "Acknowledged" after the Developer clarified their architecture and existing safeguards.

Finding 2: Non-Sequential Nonce Validation in Meta-Transaction System

Severity: Medium

Status: Resolved

Source: Forwarder.sol (_executeTransaction() / verify())

Description:

The Forwarder contract's meta-transaction implementation aims to simplify blockchain interaction for end users through EIP-712 signature validation and nonce sequence enforcement. The current implementation verifies that the recovered signer matches the request's from address and that the request's hash hasn't been previously processed. The implementation currently does not validate that req.nonce equals _nonces[req.from] before execution. This means a compromised or misconfigured relayer could submit a request with any future or out-of-order nonce, advancing the on-chain counter and invalidating all legitimate signed requests for intermediate nonces.

Since meta-transactions are designed to provide a seamless off-chain user experience that reliably maps to on-chain state, this implementation difference creates a discrepancy between user-signed transactions and chain-accepted operations. Users expecting consistent transaction execution may encounter unexpected failures, potentially impacting adoption and user experience.

Impact:

User Experience Impact:

Users depend on sequential nonces to ensure each signed request executes exactly once. When future nonces are accepted, previously signed messages become invalid, resulting in transaction failures in wallet interfaces and dApps. Users may encounter "invalid nonce" errors without clear resolution paths, potentially leading to reduced platform engagement.

System Synchronization:

Meta-transaction systems require precise alignment between off-chain counters (managed in the dApp) and on-chain nonces. Any misalignment disrupts this synchronization, affecting the core user experience benefits of meta-transactions. This divergence may impact support operations and developer integration efforts.

Code:

```
// Current: no check against on-chain nonce
// require(req.nonce == _nonces[req.from], ... );
// Enforce strict, sequential nonces to preserve UX & auditability
require(
```

```
    req.nonce == _nonces[req.from],
    "Forwarder: invalid or out-of-order nonce"
);

function verify(ForwardRequest calldata req, bytes calldata signature)
    public
    view
    returns (bool)
{
    // Recover the signer's address from the signature using EIP-712 typed data
    address signer = _hashTypedDataV4(
        keccak256(
            abi.encode(
                _TYPEHASH,
                req.from,
                req.to,
                req.value,
                req.gas,
                req.nonce,
                keccak256(req.data)
            )
        )
    ).recover(signature);

    // Verify both the signer matches the from address AND the nonce is correct
    // This prevents both signature forgery and replay attacks
    // return (signer == req.from && req.nonce == _nonces[req.from]); /// @audit commented check is prone to vulnerability
    return (signer == req.from);
}
```

Proof of Concept:

Attack Scenario:

1. Setup: A legitimate user signs meta-transactions for nonces 0, 1, 2, … in their wallet UI.
2. Malicious Injection: A compromised relayer or bridge submits a signed request for nonce 10 first.
3. On-Chain Effect: The Forwarder accepts the nonce = 10 request (no strict equality check), executes it, and sets _nonces[user] = 11.
4. Failure of Legitimate Requests: All pending requests for nonces 0…9 now fail Replay attack prevented or "invalid nonce" errors, leaving the user unable to transact until those gaps are manually bridged—or until they abandon the platform.

This attack does not steal funds directly, but cripples the user's ability to interact with the protocol. It demonstrates how lax nonce enforcement can cause irreversible gaps in the user's transaction sequence, thwarting the core value proposition of meta-transactions.

Remediation:

By restoring one-to-one alignment between off-chain signature flows and on-chain nonce state, the protocol will preserve its promise of predictable, gasless interactions, critical for onboarding and scaling to mass adoption.

Finding 3: Custom Meta-Transaction Implementation Requires Security Enhancement

Severity: Medium

Status: Resolved

Source: Cngn.sol (customSender() & meta-transaction modifiers; forwarder.sol -> EIP-712 integration)

Description:

The cNGN token contract implements a custom customSender() logic that differs from established meta-transaction patterns. The current implementation checks isTrustedForwarder(msg.sender) and extracts the last 20 bytes of calldata as the "real" sender. The current implementation could be enhanced with additional security validations:

1. Validation of calldata length or structure
2. Enforcement of domain separation
3. Protection against malicious padding or reordering

Additionally, the "deployer or forwarder" modifier is applied to critical functions (mint and burnByUser), combining the roles of gas-payer (forwarder) and protocol administrator. This design choice means a compromised relayer key could potentially gain administrative privileges, including the ability to mint tokens and burn user funds.

Impact:

1. Unauthorized Operations: A compromised relayer or malformed transaction could potentially execute operations on behalf of any user, including minting or burning tokens, or bypassing security controls.
2. Compliance Considerations: In a regulated stablecoin context, combining gas-payment roles with governance roles may impact audit trail integrity and regulatory compliance.
3. User Trust Impact: Any discrepancy between off-chain signatures and on-chain execution could affect user confidence and platform adoption.

Code:

```
/**
 * @dev Returns the sender of the transaction. If the transaction is sent
     through
 * a trusted forwarder, returns the original sender from the calldata.
 * @return signer The address of the transaction sender
 */
```

```
function customSender() internal view returns (address payable signer) {
    if (msg.data.length >= 20 && isTrustedForwarder(msg.sender)) {
        assembly {
            signer := shr(96, calldataload(sub(calldatasize(), 20)))
        }
    } else {
        signer = payable(msg.sender);
    }
    return signer;
}

/**
 * @dev Mints new tokens to a specified address.
 * The minting process requires authorization and is subject to strict controls
    :
 * 1. The signer must be authorized to mint
 * 2. The exact amount must match the pre-approved mint amount
 * 3. After minting, the authorization is automatically revoked
 *
 * @param _amount The amount of tokens to mint
 * @param _mintTo The address to mint tokens to
 * @return bool Returns true for a successful mint
 */
function mint(
    uint256 _amount,
    address _mintTo
) public virtual onlyDeployerOrForwarder nonReentrant returns (bool) {...}

/**
 * @dev Allows a user to burn their own tokens.
 * This function can only be called by the token owner or the trusted forwarder
    .
 *
 * @param _amount The amount of tokens to burn
 * @return bool Returns true for a successful burn
 */
function burnByUser(
    uint256 _amount
) public virtual onlyDeployerOrForwarder nonReentrant returns (bool) {...}
```

Remediation:

Consider implementing OpenZeppelin's ERC2771ContextUpgradeable instead of the custom implementation. This solution provides:

1. Robust handling of ABI-encoding edge cases
2. Proper data length validation and domain separation
3. Clear distinction between msg.sender and _msgSender()

Additionally:

1. Restrict sensitive functions (mint, burn, pause) to onlyOwner or specific role-based mod-

ifiers

2. Separate roles: maintain gasless UX for transfers while requiring direct key-based calls for state-changing operations
3. Implement the principle of least privilege to prevent relayer compromises from affecting administrative functions

Finding 4: Incomplete Pause Mechanism Implementation

Severity: Low

Status: Resolved

Source: Cngn.sol (burnByUser(uint256)), Cngn2.sol (transfer(address,uint256))

Description:

The transfer and burnByUser entry points currently bypass the contract's pause mechanism. While transferFrom and administrative functions implement the whenNotPaused guard, neither transfer(address,uint256) in Cngn2.sol nor burnByUser(uint256) in Cngn.sol includes this modifier. This allows token movements and self-burns to continue even when the contract has been paused for emergency response.

Impact:

During emergency situations, such as discovered vulnerabilities, regulatory requirements, or critical issues, the pause mechanism is intended to halt all token operations. The current implementation's exemptions for transfers and user-initiated burns may limit the effectiveness of emergency controls, potentially affecting the protocol's ability to respond to critical situations.

Code:

```
// In Cngn2.sol
function transfer(
    address to,
    uint256 amount
) public virtual override nonReentrant returns (bool) {
    // … missing whenNotPaused
}

// In Cngn.sol
function burnByUser(
    uint256 _amount
) public virtual onlyDeployerOrForwarder nonReentrant returns (bool) {
    _burn(_msgSender(), _amount);
    return true;
}
```

Remediation:

Consider implementing the whenNotPaused modifier on both functions to ensure consistent pause behavior:

```
- function transfer(address to, uint256 amount) public virtual override
    nonReentrant returns (bool) {
+ function transfer(address to, uint256 amount) public virtual override
    whenNotPaused nonReentrant returns (bool) {…
```

```

}

- function burnByUser(uint256 _amount) public virtual onlyDeployerOrForwarder
    nonReentrant returns (bool) {
+ function burnByUser(uint256 _amount) public virtual onlyDeployerOrForwarder
    nonReentrant whenNotPaused returns (bool) {
     _burn(_msgSender(), _amount);
     return true;
}
```

Finding 5: Inconsistent Meta-Transaction Context Implementation

Severity: Low

Status: Resolved

Source: Cngn2.sol (transfer(address,uint256), transferFrom(address,address,uint256))

Description:

While the Cngn2 upgradeable token contract implements a customSender() helper for meta-transaction signer extraction, it does not utilize this functionality in its core transfer entry points. Both transfer and transferFrom functions rely on _msgSender() (the default ContextUpgradeable value), potentially causing relayed calls to use the forwarder's address instead of the actual user's address.

Impact:

This implementation difference may affect audit trail integrity and confidence in the ERC-2771 pattern, particularly when combined with the previously identified role conflation and meta-transaction context issues.

Code:

```
/**
 * @dev Returns the sender of the transaction. If the transaction is sent
     through
 * a trusted forwarder, returns the original sender from the calldata.
 * @return signer The address of the transaction sender
 */
function customSender() internal view returns (address payable signer) {
    if (msg.data.length >= 20 && isTrustedForwarder(msg.sender)) {
        assembly {
            signer := shr(96, calldataload(sub(calldatasize(), 20)))
        }
    } else {
        signer = payable(msg.sender);
    }
    return signer;
}
```

Remediation:

Consider implementing OpenZeppelin's ERC2771ContextUpgradeable across all external entry points, including transfer and transferFrom, to ensure _msgSender() correctly resolves to the actual signer when called via a trusted forwarder. Removing the customSender() implementation would help maintain consistency:

```
address owner = _msgSender(); // resolves correctly via ERC2771Context
```

Disclaimer

This audit report ("Report") is provided by FailSafe ("Auditor") for the exclusive use of the client ("Client"). The audit scope is limited to a technical review of the Smart Contract code supplied by the Client.

While FailSafe has made every effort to identify vulnerabilities and deviations from best practices, we do not guarantee the absence of all security issues or that the Smart Contract will function as intended in every environment.

FailSafe is not liable for any losses or damages arising from the use, misuse, or inability to use the Smart Contract. This Report is not an endorsement or certification of the Smart Contract and may not be shared or reproduced without FailSafe's written consent.

The Client is solely responsible for the deployment, operation, and further testing of the Smart Contract, as well as for implementing any recommended changes. FailSafe may update this Report if new information becomes available, and the Client is encouraged to maintain ongoing security reviews.

By using this Report, the Client accepts these terms and conditions.

Appendix

Code coverage via solidity-coverage

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	19.51	17.44	17.48	20.77	
Cngn2.sol	0	0	0	0	... 401,402,403
IOperations.sol	100	100	100	100	
Multisig.sol	96	65.87	100	95.28	... 267,402,517
Operations.sol	0	0	0	0	... 209,211,213
Operations2.sol	0	0	0	0	... 208,210,212
ProxyAdmin.sol	100	100	100	100	
TestContract.sol	100	100	100	100	
cngn.sol	0	0	0	0	... 405,406,407
forwarder.sol	0	0	0	0	... 206,210,214
All files	19.51	17.44	17.48	20.77	

```
> Istanbul reports written to ./coverage/ and ./coverage.json
Error in plugin solidity-coverage: ❌ 20 test(s) failed under coverage.
```