Frank Castle Audits

cNGN Stablecoin Audit

Jan 2026

FrankCastle_cNGN_Audit_Final_Report

Frank Castle Audits

Table of Contents

About Frank Castle Audits

Frank Castle is a professional smart contract security researcher specializing in the auditing of **Rust-based smart contracts and decentralized infrastructure**, with a strong focus on the **Solana ecosystem**.

He has conducted security reviews for a wide range of high-profile and multi-million-dollar protocols, including **Lido, GMX, Pump.fun, LayerZero, Synthetix, Hydration, Perena, ORO, DUB Social**, and others.

Frank Castle has successfully completed **over 50 Solana security audits**, building a proven track record of rigor, depth, and reliability in smart contract security. He is trusted by leading audit firms and security organizations, including **Pashov Audit Group, Spearbit, Cantina, Shieldify, Zenith**, and **BailSec**.

His comprehensive experience and hands-on expertise in Rust-based ecosystems reflect a strong commitment to advancing blockchain security and promoting industry best practices.

Portfolio: https://github.com/Frankcastleauditor/public-audits
X (Twitter): https://x.com/0xcastle_chain
Telegram: https://t.me/castle_chain

Security Review

Project: cNGN Stablecoin
Commit: 73bb78cb69a5de64ba534324275e6ac0ae64ccb5
Date: January 2026

All The Findings have been Fixed by cNGN development team and reviewed by Frank Castle

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

This audit report is prepared by **Frank Castle Audits** and is provided on an “as is” basis. The findings and recommendations contained in this report are based on the reviewer’s analysis of the codebase at the time of review.

Frank Castle Audits shall not be held liable for any losses or damages arising from the use of this report or reliance on its findings.

Risk Classification

Findings are classified according to their **impact** and **likelihood**, following an industry-standard severity model.

Severity	Description
Critical	Issues that can lead to direct loss of funds, permanent protocol failure, or complete compromise of core functionality.
High	Issues that significantly affect protocol security or availability and may lead to loss of funds or major disruption under certain conditions.
Medium	Issues that can cause incorrect behavior, state corruption, or DoS scenarios but require specific conditions or privileges.
Low	Minor issues that do not pose an immediate security risk but may affect correctness, efficiency, or best practices.
Informational	Observations and recommendations that improve code quality, maintainability, or design clarity.

About cNGN

cNGN is a stablecoin protocol implemented across **Solana** and **EVM**.

Executive Summary

Severity	Count	Status
Critical	2	Fixed

Severity	Count	Status
High	2	Fixed
Medium	2	Fixed
Low	13	Fixed
Informational	1	

All the findings have been **Fixed** by cNGN development team and **reviewed** by **Frank Castle**.
``

Scope

- **EVM:** https://github.com/wrappedcbdc/stablecoin-cngn/tree/feat-cngn3/evm-contracts
- **Solana:** https://github.com/wrappedcbdc/stablecoin-cngn/tree/solana-contract/solana-contract

Commit: `73bb78cb69a5de64ba534324275e6ac0ae64ccb5`

Findings

Critical Severity

[C-01] Missing `mut` Constraint on Multisig Account Enables Infinite Replay Attacks

Severity: Critical

Description

The `validate_multisig_authorization` function in `multisig.rs:129-185` is designed to prevent replay attacks by incrementing a nonce after each successful multisig authorization:

```rust
pub fn validate_multisig_authorization(
    multisig: &mut Multisig,  // Expects mutable reference
    instructions: &AccountInfo,
    expected_message: &[u8],
) -> Result<()> {
    // ... validation logic ...

    // This prevents replay attacks
    multisig.nonce += 1;  // Line 182

    Ok(())
}
```

However, in almost all instruction contexts that call this function, the `multisig` account is **not marked as mutable** in the Anchor account constraints.

For example, in `pause.rs:9-13`

```
#[account(
    seeds = [Multisig::MULTISIG_SEED, token_config.mint.key().as_ref()],
    bump = multisig.bump
    // Missing: mut
)]
pub multisig: Account<'info, Multisig>,
```

Root Cause:

When an Anchor account constraint lacks the `mut` keyword:

1. The account is marked as **read-only** in the transaction
2. Anchor allows in-memory modifications (Rust's `&mut` reference works)
3. But Anchor does **not serialize changes back** to the account at instruction completion
4. All modifications are silently discarded

This means `multisig.nonce += 1` executes successfully in memory, but the incremented nonce is never persisted to the blockchain.

Affected Contexts:

File	Context	Line
pause.rs	`PauseMint`	9-13
admin.rs	`AddCanMint`	47-50
admin.rs	`RemoveCanMint`	127-130
admin.rs	`SetMintAmount`	198-201
admin.rs	`RemoveMintAmount`	268-271
admin.rs	`AddBlackList`	367-370
admin.rs	`RemoveBlackList`	455-458
admin.rs	`SetMintAmountMultisig`	502-505
admin.rs	`RemoveMintAmountMultisig`	568-571
admin.rs	`AddTrustedContract`	681-684
admin.rs	`RemoveTrustedContract`	754-757
admin.rs	`WhitelistInternalUser`	833-836
admin.rs	`WhitelistExternalUser`	917-920
admin.rs	`BlacklistInternalUser`	1001-1004

File	Context	Line
admin.rs	`BlacklistExternalUser`	1074-1077

Note: `UpdateMultisig` in `multisig.rs:36-41` correctly includes `mut` and is not affected.

Impact:

An attacker who obtains a valid set of multisig signatures for any operation can replay that transaction indefinitely. The attack flow:

1. Multisig owners sign a message with `nonce = N` (e.g., to add a user to the blacklist)
2. Transaction executes successfully, but nonce remains `N` on-chain
3. Attacker replays the exact same transaction
4. Since nonce is still `N`, the same signatures remain valid
5. This can be repeated infinitely

This completely defeats the replay protection mechanism and allows:

- Repeated pause/unpause of minting
- Repeated addition/removal of minters, blacklist entries, whitelisted users
- Repeated modification of trusted contracts
- Any multisig-protected action to be executed multiple times with a single set of signatures

Recommendation

Add the `mut` keyword to all multisig account constraints that call `validate_multisig_authorization`:

```
#[account(
    mut,  // Add this
    seeds = [Multisig::MULTISIG_SEED, token_config.mint.key().as_ref()],
    bump = multisig.bump
)]
pub multisig: Account<'info, Multisig>,
```

[C-02] Unvalidated PDA Derivation Allows Cross-Token Access Control Bypass

Severity: Critical

Description

in the `MintTokens`, the`mint_authority`, `blacklist`, `can_mint`, and `trusted_contracts` accounts are not derived from or constrained to the specific `mint` account. This allows an attacker to substitute these accounts from a different token deployment to bypass access controls.

The `MintTokens` struct validates that the `mint` matches the `token_config` and that the `token_account` corresponds to the correct mint:

```
// mint.rs:12-29
#[account(
    constraint = !token_config.mint_paused @ ErrorCode::MintingPaused)]
pub token_config: Account<'info, TokenConfig>,

#[account(
    mut,
    constraint = mint.key() == token_config.mint @ ErrorCode::MintMismatch,
)]
pub mint: InterfaceAccount<'info, Mint>,

#[account(
    mut,
    constraint = token_account.mint == mint.key() @ ErrorCode::MintMismatch,
)]
pub token_account: InterfaceAccount<'info, TokenAccount>,
```

However, the security-critical accounts lack any derivation or constraint linking them to the mint:

```
// mint.rs:31-38
#[account(mut)]
pub blacklist: Account<'info, BlackList>,

#[account(mut)]
pub can_mint: Account<'info, CanMint>,

#[account(mut)]
pub trusted_contracts: Account<'info, TrustedContracts>,
```

The handler relies on these accounts for critical access control checks:

```
// mint.rs:48-67
// Check if signer is blacklisted
require!(
    !ctx.accounts.blacklist.is_blacklisted(&signer),
    ErrorCode::SignerBlacklisted
);

// Check if receiver is blacklisted
require!(
    !ctx.accounts.blacklist.is_blacklisted(&mint_to),
    ErrorCode::ReceiverBlacklisted
);

// Check if signer is authorized to mint
require!(
    ctx.accounts.can_mint.can_mint(&signer),
    ErrorCode::MinterNotAuthorized
);

// Check if mint amount matches the allowed amount
let allowed_amount = ctx.accounts.can_mint.get_mint_amount(&signer)?;
require!(amount == allowed_amount, ErrorCode::InvalidMintAmount);
```

Attack Scenario:

1. Attacker initialise a separate token instance thought calling initialise functions of the cNGN program and initialise multisig to be the owners where they control the `can_mint` list and every parameter related to controls over that different mint account
2. Attacker adds themselves to that separate `can_mint` account with a large mint amount
3. Attacker calls `mint` on the target token (e.g., cNGN) but passes their controlled `can_mint` account
4. All checks pass because the `can_mint` account is not validated against the target mint
5. Attacker successfully mints tokens on the target token without proper authorization

Impact: This vulnerability allows complete bypass of:

- **Blacklist enforcement**: Blacklisted users can mint by providing a different blacklist account
- **Minter authorization**: Unauthorized users can mint by providing a `can_mint` account where they are authorized
- **Mint amount limits**: Users can mint arbitrary amounts by controlling the `can_mint` account's allowed amounts

This results in unauthorized token minting, rendering the entire access control system ineffective.

Recommendation

use the same deprivation logic as in `pub struct AddCanMint<'info>`

```
    #[account(
        seeds = [BLACK_LIST_SEED, token_config.mint.as_ref()],
        bump,
    )]
    pub blacklist: Account<'info, BlackList>,

    #[account(
        mut,
        seeds = [CAN_MINT_SEED, token_config.mint.as_ref()],
        bump,
    )]
    pub can_mint: Account<'info, CanMint>,

    #[account(
        seeds = [TRUSTED_CONTRACTS_SEED, token_config.mint.as_ref()],
        bump,
    )]
    pub trusted_contracts: Account<'info, TrustedContracts>,

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

High Severity

[H-01] Missing Amount Parameter in Multisig Message Hash Allows Unauthorized Mint Amount Manipulation

Severity: High

Description

in `set_mint_amount` function `amount` parameter is not included in the message hash used for multisig signature verification. This allows an attacker to manipulate the mint amount while using a valid multisig signature giving himself more amounts than initially intended by the admins

The `set_mint_amount` function accepts three key parameters:

```
// lib.rs:66-68
pub fn set_mint_amount(ctx: Context<SetMintAmount>, user: Pubkey, amount:
u64) -> Result<()> {
    instructions::admin::set_mint_amount_handler(ctx, user, amount)
}
```

However, when constructing the message for multisig validation, only the `can_mint_account`, `user`, and `nonce` are included—the `amount` is omitted:

```
// multisig.rs:237-250
pub fn build_set_mint_amount_message(
    can_mint_account: &Pubkey,
    user: &Pubkey,
    nonce: u64,
) -> Vec<u8> {
    let mut hasher = Sha256::new();

    hasher.update(b"SET_MINT_AMOUNT");
    hasher.update(can_mint_account.as_ref());
    hasher.update(user.as_ref());
    hasher.update(&nonce.to_le_bytes());

    hasher.finalize().to_vec()
}
```

This creates a scenario where:

1. Multisig signers authorize setting a mint amount for a specific user (e.g., 10 tokens)
2. The signed message does not commit to the actual amount value
3. Anyone who observes this valid signature can submit the transaction with an arbitrary amount (e.g., 1,000,000 tokens)
4. The signature verification passes because the amount was never part of the signed data

Impact: This vulnerability allows bypass of the intended control over mint amounts.

Recommendation

Include the `amount` parameter in the message hash to ensure multisig signers explicitly authorize the specific amount:

```
pub fn build_set_mint_amount_message(
    can_mint_account: &Pubkey,
    user: &Pubkey,
    amount: u64,  // Add amount parameter
    nonce: u64,
) -> Vec<u8> {
    let mut hasher = Sha256::new();

    hasher.update(b"SET_MINT_AMOUNT");
    hasher.update(can_mint_account.as_ref());
    hasher.update(user.as_ref());
    hasher.update(&amount.to_le_bytes());  // Include amount in hash
    hasher.update(&nonce.to_le_bytes());

    hasher.finalize().to_vec()
}
```

Update the call site in `admin.rs` accordingly:

```
let message =
    build_set_mint_amount_message(&ctx.accounts.can_mint.key(), &user,
amount, multisig.nonce);
```

[H-02] Unrestricted Multisig Initialization Enables Admin Privilege Takeover

Severity: High

Description

in the `initialize_multisig` function, any user can invoke the function to create a multisig account and immediately gain control over the token configuration. The function transfers admin ownership to the newly created multisig.

The `initialize_multisig` function accepts arbitrary owners and threshold parameters without any access control:

```
// lib.rs:31-37
pub fn initialize_multisig(
    ctx: Context<InitializeMultisig>,
    owners: Vec<Pubkey>,
    threshold: u8,
) -> Result<()> {
    instructions::multisig::initialize_multisig_handler(ctx, owners,
threshold)
}
```

The handler performs basic validation on the owners list and threshold but lacks any authorization check before transferring admin privileges:

```
// multisig.rs:55-82
pub fn initialize_multisig_handler(
    ctx: Context<InitializeMultisig>,
    owners: Vec<Pubkey>,
    threshold: u8,
) -> Result<()> {
    require!(
        owners.len() <= Multisig::MAX_OWNERS,
        ErrorCode::TooManyOwners
    );

    require!(!owners.is_empty(), ErrorCode::NoOwnersProvided);
    let token_config = &mut ctx.accounts.token_config;
    // Check for duplicate owners
    let mut unique_owners = BTreeSet::new();
    for owner in &owners {
        require!(unique_owners.insert(owner), ErrorCode::DuplicateOwners);
    }

    Multisig::assert_valid_threshold(owners.len(), threshold)?;

    let multisig = &mut ctx.accounts.multisig;
    multisig.owners = owners;
    multisig.threshold = threshold;
    multisig.nonce = 0;
    multisig.bump = ctx.bumps.multisig;
    token_config.admin = multisig.key();  // Immediate ownership transfer
    Ok(())
}
```

Attack Scenario:

1. deployer calls `initialize` to create a new cNGN token
2. Attacker observes the new token_config account
3. Attacker calls `initialize_multisig` with their own wallet as the sole owner and threshold of 1
4. The function creates a multisig controlled entirely by the attacker
5. Line 80 executes: `token_config.admin = multisig.key()` — transferring all admin rights to the attacker
6. Attacker now has full administrative control over the token

Impact: This allows takeover of the token system. An attacker gaining admin control can:

- **Pause/unpause minting** at will, disrupting token operations
- **Modify the blacklist**, removing sanctions enforcement or blacklisting legitimate users
- **Grant minting privileges** to arbitrary addresses with unlimited amounts
- **Modify trusted contracts**, potentially enabling malicious integrations
- **Brick the token** by setting configurations that prevent normal operation

Recommendation

Implement strict access control to ensure only authorized parties can initialize the multisig:

Option 1: Restrict to Current Admin

```
#[derive(Accounts)]
pub struct InitializeMultisig<'info> {
    #[account(
        mut,
        constraint = token_config.admin == current_admin.key() @
ErrorCode::Unauthorized
```

```
    )]
    pub token_config: Account<'info, TokenConfig>,

    pub current_admin: Signer<'info>,

    // ... other accounts
}
```

Medium Severity

[M-01] Missing Redemption Flow in `transferFrom` Leads to Unburnt Tokens

Severity: Medium

Description

In `cngn3.sol:109-147`, the `transfer` function implements a special redemption flow: when an external whitelisted sender transfers to an internal whitelisted recipient, the tokens are transferred and then immediately burnt:

```
// transfer() - Lines 126-132
if (
    IAdmin(adminOperationsContract).isInternalUserWhitelisted(to) &&
    IAdmin(adminOperationsContract).isExternalSenderWhitelisted(owner)
) {
    _transfer(owner, to, amount);
    _burn(to, amount);  // Tokens are burnt for redemption
}
```

However, in `cngn3.sol:157-181`, the `transferFrom` function performs a plain transfer with no redemption logic:

```
// transferFrom() - Lines 178-179
_spendAllowance(from, spender, amount);
_transfer(from, to, amount);
// No whitelist check, no burn
```

If the off-chain redemption system is triggered by `Transfer` events from an external whitelisted address to an internal whitelisted address, both `transfer` and `transferFrom` will emit the same `Transfer` event. The off-chain system would process the redemption in both cases, but only `transfer` actually burns the tokens on-chain.

This creates a discrepancy: when `transferFrom` is used, the off-chain system redeems (e.g., releases fiat), but the tokens remain in the internal whitelisted account's balance — they are never burnt. This may allow double-spending: the redeemed value is released off-chain while the tokens still exist on-chain and can be transferred or redeemed again.

Recommendation

Apply the same redemption logic in `transferFrom`:

```
function transferFrom(address from, address to, uint256 amount) ... {
    // ... blacklist checks ...

    _spendAllowance(from, spender, amount);

    if (
        IAdmin(adminOperationsContract).isInternalUserWhitelisted(to) &&
        IAdmin(adminOperationsContract).isExternalSenderWhitelisted(from)
    ) {
        _transfer(from, to, amount);
        _burn(to, amount);
    } else {
        _transfer(from, to, amount);
    }

    return true;
}
```

[M-02] Unrestricted Initialization Allows Unauthorized Token Deployments Under cNGN Program

Severity: Medium

Description

in the `initialize` function, any user can invoke the function to deploy new tokens using the cNGN program's infrastructure. Allowing unrestricted token creation through this program may create legal and reputational risks.

The `initialize` function accepts parameters for creating a new token with no access control restrictions:

```
// lib.rs:18-26
pub fn initialize(
    ctx: Context<Initialize>,
    name: String,
    symbol: String,
    uri: String,
    decimals: u8,
) -> Result<()> {
    instructions::initialize::initialize_handler(ctx, name, symbol, uri,
decimals)
}
```

Recommendation

Implement an authorized deployer mechanism that restricts who can initialize new tokens

Low Severity

[L-01] 63-64 Gas Griefing Vulnerability in Meta-Transaction Execution

Severity: Low

Description

In `forwarder.sol:148-150`, the external call does not specify a gas limit:

```
(bool success, bytes memory returndata) = req.to.call{value: req.value}(
    abi.encodePacked(req.data, req.from)
);
```

The `ForwardRequest` struct includes a `gas` field (line 27), but it's never used. Due to the EVM's 63/64 rule, a malicious or negligent relayer can provide just enough gas for the forwarder's logic to complete while the inner call revert our of gas.

Example:

1. User signs a meta-transaction requesting 500k gas
2. Relayer submits with ~100k gas total
3. Forwarder logic consumes ~30k gas, forwards ~44k to the call (63/64 of remaining ~70k)
4. user call fails due to insufficient gas
5. Transaction succeeds (no revert), nonce is incremented, user's transaction is "used up"
   fowarder doesn't have to be malicious, but this case can happen silently

Recommendation

Include the request gas in the call and implement a 63/64 protection compared to the request gas value

```
require(gasleft() >= (req.gas * 64) / 63, "Insufficient gas for requested
execution");

(bool success, bytes memory returndata) = req.to.call{gas: req.gas, value:
req.value}(
    abi.encodePacked(req.data, req.from)
);
```

[L-02] Changing Admin Operations Contract Address Invalidates Blacklist and Access Control State

Severity: Low

Description

In `cngn3.sol:66-76`, the `updateAdminOperationsAddress` function allows the owner to change the admin operations contract :

```
function updateAdminOperationsAddress(
    address _newAdmin
) public virtual onlyOwner returns (bool) {
    require(
        _newAdmin != address(0),
        "New admin operations contract address cannot be zero"
    );
    emit UpdateAdminOperations(adminOperationsContract, _newAdmin);
    adminOperationsContract = _newAdmin;
    return true;
}
```

The token contract relies on the admin operations contract for access control data:

- **Blacklist** - `isBlackListed()` checks (lines 117, 121, 166, 170, 174, 200, 204, 237, 258)
- **Whitelist** - `isInternalUserWhitelisted()`, `isExternalSenderWhitelisted()` (lines 127, 128)
- **Minting authorization** - `canMint()`, `mintAmount()`, (lines 208, 212)

Root Cause:

When the admin operations contract address is changed to a new contract, the new contract has **fresh/empty storage**. This means:

1. All previously blacklisted users become **unblacklisted**
2. special roles whitelisted lose their whitelist status
3. All minting authorizations are reset

Even if the new contract is a fresh deployment of the same code, it won't have the same state (blacklist entries, whitelist entries, minting permissions) from the old contract.

Impact:

- **Blacklist Bypass (High)**: Malicious actors who were blacklisted can immediately transfer, mint, and operate freely after the admin operations address is updated
- **Compliance Violation**: Regulatory requirements for blocking sanctioned addresses are circumvented
- **Authorization Reset**: All carefully configured minting permissions are lost

The same issue exists in `forwarder.sol:52-58`:

```
function updateAdminOperationsAddress(
    address newAdmin
) external onlyOwner returns (bool) {
    adminOperationsContract = newAdmin;
    emit AdminOperationsAddressUpdated(newAdmin);
    return true;
}
```

This affects blacklist checks at lines 123 and 128, and forwarder authorization at line 118.

Recommendation

Option 1 (Recommended): Make the admin operations contract reference immutable in both contracts. Since the admin operations contract itself is upgradeable (using a proxy pattern), all logic upgrades should be performed through its proxy mechanism rather than changing the reference in the token contract.

```
address public immutable adminOperationsContract;

constructor(address _adminOperationsContract) {
    adminOperationsContract = _adminOperationsContract;
}
```

[L-03] Duplicate Error Messages Make Debugging Difficult

Severity: Low

Description

In `errors.rs`, two different error codes share the exact same message:

```
// Lines 93-94
#[msg("Attempting to mint more than allowed")]
InvalidMintAmount,

// Lines 156-157
#[msg("Attempting to mint more than allowed")]
InvalidTokenAccount,
```

When either error is thrown, developers cannot distinguish which condition failed without examining the code path, making debugging more difficult.

Recommendation

Provide unique, descriptive messages for each error:

```
#[msg("Mint amount does not match the allowed amount")]
InvalidMintAmount,

#[msg("Invalid token account provided")]
InvalidTokenAccount,
```

[L-04] Inconsistent Handling of Duplicate/Non-Existent Entries Across Admin Functions

Severity: Low

Description

Admin functions handle "already exists" and "not found" cases inconsistently:

Function	Condition	Behavior
`add_blacklist_handler`	User already blacklisted	Returns `Error`
`add_trusted_contract_handler`	Contract already trusted	Returns `Ok(())` silently
`whitelist_internal_user_handler`	User already whitelisted	Returns `Ok(())` silently
`remove_blacklist_handler`	User not blacklisted	Returns `Ok(())` silently

Examples from `admin.rs`

```
// Line 77-79: add_blacklist ERRORS on duplicate
if blacklist.is_blacklisted(&user) {
    return Err(ErrorCode::UserBlacklisted.into());
}

// Line 713-715: add_trusted_contract SILENTLY SUCCEEDS on duplicate
if trusted_contracts.is_trusted_contract(&contract) {
    return Ok(());
}

// Line 481-483: remove_blacklist SILENTLY SUCCEEDS when not found
if !blacklist.is_blacklisted(&user) {
    return Ok(());
}
```

silent successes still consume multisig nonces (when the `mut` fix is applied), wasting valid signatures on no-op operations and confusing multisig owners

Recommendation

Standardize the behavior across all admin functions with your intended logic:

- either revert on all no-op operations
- silently succeed on all no-op operations
  Hence the owners of the multisig will expect what `nonce` they should sign on.

[L-05] Missing `whenNotPaused` Modifier on `executeByBridge` Function

Severity: Low

Description

In `forwarder.sol:172-183`, the `executeByBridge` function lacks the `whenNotPaused` modifier, while the similar `execute` function includes it:

```
// Line 157-169: execute() has whenNotPaused
function execute(...)
    external payable
    onlyOwner
    whenNotPaused
    nonReentrant
    returns (bool, bytes memory)

// Line 172-183: executeByBridge() misses whenNotPaused
function executeByBridge(...)
    external payable
    onlyAuthorizedBridge
    nonReentrant
    returns (bool, bytes memory)
```

When the contract is paused via `pause()`, the owner's `execute` function is blocked, but authorized bridges can still execute meta-transactions through `executeByBridge`.

This defeats the purpose of the pause mechanism, which is typically used during emergencies (security incidents, discovered vulnerabilities, regulatory requirements).

Recommendation

Add the `whenNotPaused` modifier to `executeByBridge`:

```
function executeByBridge(
    ForwardRequest calldata req,
    bytes calldata signature
)
    external
    payable
    onlyAuthorizedBridge
    whenNotPaused
    nonReentrant
    returns (bool, bytes memory)
{
    return _executeTransaction(req, signature);
}
```

[L-06] Missing Pause State Parameter in Multisig Message Hash

Severity: Low

Description

A consistency issue exists in the `pause_minting` instruction where the `pause_mint` boolean parameter is not included in the message hash used for multisig signature verification.

The `pause_minting` function accepts a boolean parameter to set the pause state:

```
// lib.rs:54-56
pub fn pause_minting(ctx: Context<PauseMint>, pause_mint: bool) ->
Result<()> {
    instructions::pause::pause_mint_handler(ctx, pause_mint)
}
```

The message hash only includes the token_config key and nonce, omitting the intended pause state:

```
// pause.rs:34
let message = build_pause_mint_message(&ctx.accounts.token_config.key(),
multisig.nonce);
```

Recommendation

For consistency and defense-in-depth, include the `pause_mint` parameter in the message hash:

```
pub fn build_pause_mint_message(
    token_config: &Pubkey,
    pause_mint: bool,  // Add pause state parameter
    nonce: u64,
) -> Vec<u8> {
    let mut hasher = Sha256::new();

    hasher.update(b"PAUSE_MINT");
    hasher.update(token_config.as_ref());
    hasher.update(&[pause_mint as u8]);  // Include pause state in hash
    hasher.update(&nonce.to_le_bytes());

    hasher.finalize().to_vec()
}
```

and update the calling function `pause_mint_handler()` to call `build_pause_mint_message()` including `pause_mint`

[L-07] Missing Program ID in Multisig Message Signatures

Severity: Low

Description

The multisig message builders in multisig.rs do not include the program ID in the signed messages:

```rust
pub fn build_add_can_mint_message(...) -> Vec<u8> {
    let mut hasher = Sha256::new();
    hasher.update(b"ADD_CAN_MINT");
    hasher.update(can_mint_account.as_ref());
    // No program ID included
    hasher.update(user.as_ref());
    hasher.update(&nonce.to_le_bytes());
    hasher.finalize().to_vec()
}
```

While account addresses provide some uniqueness per deployment, including the program ID would provide defense-in-depth against theoretical cross-program signature reuse.

Recommendation

Include the program ID in all message hashes for additional domain separation:

```rust
hasher.update(crate::id().as_ref());
```

[L-08] No Events Emitted for Multisig Configuration Changes

Severity: Low

Description

In multisig.rs, neither `initialize_multisig_handler` nor `update_multisig_handler` emit events to record configuration changes.

This makes it difficult to:

- Track multisig ownership changes off-chain
- Audit historical configuration modifications
- Build monitoring/alerting systems for governance changes

Recommendation

Add events for multisig operations:

```
emit!(MultisigInitializedEvent {
    multisig: multisig.key(),
    owners: owners.clone(),
    threshold,
});

emit!(MultisigUpdatedEvent {
    multisig: multisig.key(),
    old_owners: old_owners,
    new_owners: new_owners.clone(),
    old_threshold: old_threshold,
    new_threshold: new_threshold,
});
```

[L-09] Redundant Authorization Check in change_admin

Severity: Low

Description

In `admin.rs:1157-1173`, the same authorization check is performed twice:

```
// First check (lines 1157-1161)
require_keys_eq!(
    ctx.accounts.authority.key(),
    token_config.admin,
    ErrorCode::Unauthorized
);

// Second check (lines 1171-1173) - REDUNDANT
if token_config.admin != ctx.accounts.authority.key() {
    return Err(ErrorCode::Unauthorized.into());
}
```

This is redundant code that adds unnecessary compute cost.

Recommendation

Remove the second check as the first `require_keys_eq!` already validates the same condition.

[L-10] Redundant Blacklist Checks in `transfer` Function

Severity: Low

Description

In `cngn3.sol:134-142`, the `else` branch of the `transfer` function performs blacklist checks that were already executed earlier in the same function:

```
function transfer(address to, uint256 amount) public virtual override
whenNotPaused nonReentrant returns (bool) {
    address owner = _msgSender();

    // Lines 116-123: First check (always executed)
    require(!IAdmin(adminOperationsContract).isBlackListed(owner), "Sender
is blacklisted");
    require(!IAdmin(adminOperationsContract).isBlackListed(to), "Recipient
is blacklisted");

    if (...) {
        // Redemption flow
    } else {
        // Lines 135-142: Duplicate check (redundant)
        require(!IAdmin(adminOperationsContract).isBlackListed(owner),
"Sender is blacklisted");
        require(!IAdmin(adminOperationsContract).isBlackListed(to),
"Recipient is blacklisted");
        _transfer(owner, to, amount);
    }
}
```

Lines 116-123 already check both `owner` and `to` against the blacklist **before** the `if/else` branch. The checks at lines 135-142 are unreachable in a failing state - if either address were blacklisted, the function would have already reverted.

This wastes gas on two redundant external calls to `adminOperationsContract.isBlackListed()` for every standard transfer.

Recommendation

Remove the duplicate checks in the `else` branch:

```
} else {
    _transfer(owner, to, amount);
}
```

[L-11] Syntax Error in Mint Function Blacklist Check

Severity: Low

Description

In `cngn3.sol:199-202`, the `require` statement uses a logical OR operator (`||`) instead of a comma to separate the condition from the error message:

```
require(
    !IAdmin(adminOperationsContract).isBlackListed(sender) ||  // ← should
be comma
    "Signer is blacklisted"
);
```

Solidity's `||` operator requires both operands to be `bool`. A string literal is not `bool`, so this is a compilation error.

Compare with the correct syntax on line 203-205 immediately below:

```
require(
    !IAdmin(adminOperationsContract).isBlackListed(_mintTo),  // ← correct
comma
    "receiver is blacklisted"
);
```

Recommendation

Replace the `||` with a comma:

```
require(
    !IAdmin(adminOperationsContract).isBlackListed(sender),
    "Signer is blacklisted"
);
```

[L-12] TokensMintedEvent Emits Token Account Address Instead of Recipient Owner

Severity: Low

Description

In `mint.rs:107-112`, the `TokensMintedEvent` emits the token account (ATA) address in the `to` field rather than the actual recipient's wallet address:

```
let mint_to = ctx.accounts.token_account.owner;  // Line 45 - actual
recipient

// ...

emit!(TokensMintedEvent {
    mint: ctx.accounts.mint.key(),
    to: ctx.accounts.token_account.key(),  // Emits ATA address, not owner!
    amount,
});
```

The variable `mint_to` correctly holds `token_account.owner` (the recipient's wallet), but the event emits `token_account.key()` (the Associated Token Account address).

Off-chain systems relying on these events for tracking mints would record the wrong address, potentially causing accounting discrepancies or failed lookups.

Recommendation

Update the event to emit the recipient's wallet address:

```
emit!(TokensMintedEvent {
    mint: ctx.accounts.mint.key(),
    to: mint_to,  // Use the owner, not the ATA
    amount,
});
```

[L-13] Transfer Pause Message Builder Exists But No Instruction Uses It

Severity: Low

Description

The codebase includes `build_pause_transfer_message` in `multisig.rs:436-444` and `transfer_paused` field in `TokenConfig`, but no instruction exists to actually pause transfers.

```
pub fn build_pause_transfer_message(token_config_account: &Pubkey, nonce: u64) -> Vec<u8> {
    let mut hasher = Sha256::new();
    hasher.update(b"PAUSE_TRANSFER");
    hasher.update(token_config_account.as_ref());
    hasher.update(&nonce.to_le_bytes());
    hasher.finalize().to_vec()
}
```

This is dead code that suggests incomplete functionality.

Recommendation

Either implement the `pause_transfer` instruction or remove the unused code to avoid confusion.

Informational

- Uninitialized fields in MintAuthority and CanForward
- InterfaceAccount accepts both token programs while token program in initialise is Token2022 which accepts only token2022
- `token_program` in `struct Initialize<'info>` is redundant and not used

- No minimum number of threshold/owners enforcement during multisign initialise/update mechanisms. accepting a 1 owner with one threshold contradicting the multisign design
- Unused functions logic function `add_can_forward()`, `remove_can_forward()`, `whitelist_external_user()`, `blacklist_external_user()`, `whitelist_internal_user()`, `blacklist_internal_user()`, `add_trusted_contract()`, `remove_trusted_contract()` and `initialize_third_handler()`
- `change_admin()` is unused since the admin will need to initialize multisig to operate any function in the program. hence the presence of the function is not necessary

Contact

For questions regarding this audit or follow-up security reviews, please contact:

- **Portfolio:** https://github.com/Frankcastleauditor/public-audits
- **X (Twitter):** https://x.com/0xcastle_chain
- **Telegram:** https://t.me/castle_chain