ClarityAlliance

USDCx SECURITY REVIEW

Conducted by:
KRISTIAN APOSTOLOV, SILVEROLOGIST

JANUARY 21st, 2026

Contents

1. About Clarity Alliance

Clarity Alliance is a team of expert whitehat hackers specialising in securing protocols on Stacks.

They have disclosed vulnerabilities that have saved millions in live TVL and conducted thorough reviews for some of the largest projects across the Stacks ecosystem.

Learn more about Clarity Alliance at clarityalliance.org.

2. Disclaimer

This report is not, and should not be considered, an endorsement or disapproval of any project, team, product, or asset, nor does it reflect on their economics, value, business model, or legal compliance. It does not provide any warranty regarding the absolute bug-free nature or functionality of the analyzed technology.

Nothing in this report should be used to make investment or participation decisions. It does not constitute investment advice. Instead, it reflects an extensive assessment process intended to help clients improve code quality and reduce the inherent risks associated with cryptographic tokens and blockchain systems.

Blockchain technology presents ongoing and significant risk, and each company or individual remains responsible for their own due diligence and security posture. Clarity Alliance aims to reduce attack vectors and technological uncertainty but does not guarantee the security or performance of any system we review.

All assessment services are subject to dependencies and active development. Your access to and use of any services, reports, or materials is at your sole risk on an as-is and as-available basis.

Cryptographic tokens are emergent technologies with high technical uncertainty, and assessment results may include false positives, false negatives, or other unpredictable outcomes. Smart contracts may depend on multiple external parties, remain vulnerable to internal or external exploitation, and may carry elevated risks if owner privileges remain active. Accordingly, Clarity Alliance does not guarantee the explicit security of any audited smart contract, regardless of the reported verdict.

3. Introduction

A time-boxed security review of USDCx protocol, where Clarity Alliance reviewed the scope and provided insights on improving the protocol.

4. About USDCx Protocol

USDCx is a 1:1 USDC-backed stablecoin issued through Circle xReserve and native to Stacks.

Stacks now has a fully USDC-backed stablecoin that plugs directly into Circle's multichain ecosystem and brings stable, interoperable dollar liquidity to Bitcoin's leading Layer 2.

What is USDCx?

USDCx is a 1:1 USDC-backed stablecoin issued through Circle xReserve and native to Stacks. It will exist as a SIP-010 token on Stacks.

Circle's xReserve provides cryptographic attestations for deposits and minting, while Circle Gateway and CCTP handle cross-chain movement. The result is USDC on Stacks without third-party bridges, wrapped assets, or fragmented liquidity.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

Scope

The scope of this security assessment focused on the core components of the USDCx cross-chain protocol, specifically the Clarity smart contracts deployed on Stacks and the off-chain Attestor backend service responsible for burn attestation and Circle API interactions.

The following contracts, located in the USDCx repository, were in scope of the security review:

- `clarity/contracts/usdcx-token.clar`
- `clarity/contracts/usdcx-v1.clar`
- `attestor/src/services/burn-signer-service.ts`
- `attestor/src/services/circle-api-service.ts`
- `attestor/src/burn-poller.ts`
- `attestor/src/config.ts`
- `attestor/src/deposit-poller.ts`
- `attestor/src/relay.ts`

Initial Commit Reviewed

`3e20bf0cf97e22400fe2146ad2e23b761b0568c4`

Final Commit After Remediations

`69124961ac3a9d42b0d4eedd94a82e0ca27e972f`

7. Executive Summary

Over the course of the security review, Kristian Apostolov, Silverologist engaged with - to review USDCx Protocol. In this period of time a total of **19** issues were uncovered.

Protocol Summary

Protocol Name	USDCx
Date	January 21st, 2026

Findings Count

Severity	Amount
High	1
Medium	3
Low	9
QA	6
Total Findings	**19**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Lack of Support for Multiple Burns Within a Single Transaction	High	Resolved
[M-01]	Used Nonces Are Not Migratable To New Implementation Contract	Medium	Acknowledged
[M-02]	Infinite Loop When Fetching Page With Zero Attestations	Medium	Resolved
[M-03]	Expired Withdrawals Are Not Retried	Medium	Resolved
[L-01]	Governance Can Brick Itself	Low	Acknowledged
[L-02]	Burn Relies on tx-sender for Authentication	Low	Acknowledged
[L-03]	Burns Are Processed From Only One Minter Per Burn Block	Low	Resolved
[L-04]	Hardcoded Zero Relay Fee Leads to Incremental Losses	Low	Acknowledged
[L-05]	Confirmation Blocks Configuration Ignored	Low	Resolved
[L-06]	Mutating Token Metadata Is Problematic	Low	Acknowledged
[L-07]	Burn Poller Processes TenureChange Blocks	Low	Acknowledged
[L-08]	Unfiltered Transaction Event Receipt Looping	Low	Acknowledged

[L-09]	Missing Filter Causes Increasing Deposit Processing Delays	Low	Acknowledged
[QA-01]	Missing Event Emissions in Setter Functions	QA	Acknowledged
[QA-02]	Redundant Function Definitions	QA	Acknowledged
[QA-03]	Isolate Roles Into Constants	QA	Acknowledged
[QA-04]	Incorrect Event Field Value	QA	Acknowledged
[QA-05]	Miners Can Capture All Fees	QA	Acknowledged
[QA-06]	Inconsistent Constant Naming Format	QA	Acknowledged

8. Findings

8.1. High Findings

[H-01] Lack of Support for Multiple Burns Within a Single Transaction

Location:

- burn-attestor-service.ts#L162
- circle-api-service.ts#L223

Description
The burn poller processes burn events individually. For each event, it separately calls the `prepareWithdrawal` and `submitBurnIntent` APIs.

However, the Circle API permits only one active withdrawal per `burnTxId` (the Stacks transaction ID that contains the burn event). The xReserve documentation states that if multiple withdrawals are submitted with the same `burnTxId`, the API returns a 409/400 error indicating a conflict with an existing active withdrawal.

The system interprets this response as an `AlreadyAttestedError` and skips the corresponding burn event, even when the attestation refers to a different burn within the same transaction.

Consequently, only the first burn event in a transaction is successfully submitted to the xReserve API, while all subsequent burn events in that transaction are ignored.

Recommendation

Update the burn poller to process all burn events in a transaction together as a batch.

When a transaction containing multiple valid burn events is detected, all events should be sent to the `prepare-withdrawal` endpoint in a single request. The endpoint will return the prepared burn intents, with each intent placed in a separate batch and assigned its own message hash to sign.

To avoid `burnTxId` conflicts and comply with the `withdraw` endpoint's maximum batch size of 10, merge the intents into appropriately sized batches locally. For each batch, compute a single message hash to sign, submit the resulting data to the `withdraw` endpoint, and wait for the batch to reach a terminal state before processing the next one.

8.2. Medium Findings

[M-01] Used Nonces Are Not Migratable To New Implementation Contract

Location:

- usdcx-v1.clar

Description The system is designed to support future versions of the `usdcx-v1` contract by allowing new versions to be added as authorized USDCx minters.

During an upgrade, the previous minter can be deprecated by clearing the `circle-attestors` mapping. However, there is currently no straightforward mechanism to migrate previously used nonces from the old contract to the new one. Without such a mechanism, deposit intents that were already processed could be replayed against the new contract.

Recommendation

Consider moving `used-nonces` to either a separate storage contract that persists across system implementation upgrades, or directly into `usdcx-token`. The latter would also require modifying `protocol-mint` to accept a nonce as an argument.

[M-02] Infinite Loop When Fetching Page With Zero Attestations

Location:

- deposit-poller.ts#L96

Description The deposit poller invokes `pollOnce` on a fixed interval. After each execution, it advances its cursor using the `nextPageCursor` from the response—except when the fetched page contains zero attestations. In that case, the cursor is left unchanged, causing subsequent polls to refetch the same page:

```
const response = yield* circleService.fetchAttestations({
      remoteDomain: yield* remoteDomainConfig,
      pageSize: yield* pageSizeConfig,
@>    pageAfter: state.cursor,
    });

    const attestationCount = response.attestations.length;

    if (attestationCount === 0) {
      yield* Effect.logDebug("No new attestations found");
@>    return;
    }

    yield* Effect.logDebug(`Found ${attestationCount} new attestation(s)`);

    let newProcessed = 0;

    for (const attestation of response.attestations) {
      const result = yield* handleAttestation(attestation);
      if (Option.isSome(result)) newProcessed++;
    }

    // Update state with new cursor and count
    yield* Ref.update(stateRef, (s) => ({
@>    cursor: response.nextPageCursor,
      processedCount: s.processedCount + newProcessed,
    }));
```

Based on observed testnet xReserve behavior, the cursor for the last page becomes stale whenever a new deposit attestation is added. This stale cursor does not return newly added attestations. Instead, the client must refetch from the first page and then paginate forward to reach the updated last page.

This results in the following failure scenario:

- `fetchAttestations` is called with a non-null cursor pointing to the last page.
- The response contains zero attestations.

Because the cursor is not updated in this case, the next poll repeats the same request with the same stale cursor and again receives an empty result. This creates an infinite loop in the deposit poller, preventing new deposit attestations from ever being processed.

Recommendation

Remove the early return when zero attestations are fetched so that the cursor is always updated.

[M-03] Expired Withdrawals Are Not Retried

Location:

- burn-poller.ts

Description For each burn event, the burn poller has the attestor prepare the withdrawal, sign it, and submit it to the xReserve API.

However, according to the xReserve documentation, a successful submission does not guarantee that the withdrawal will ultimately be processed successfully.

To track progress, the API exposes the `/v1/withdrawal/{withdrawalId}` endpoint, which returns the withdrawal's status. The possible terminal statuses are:

```
'finalized': Transaction finalized successfully (terminal state)
'expired': Attestation expired before being used
  (terminal state, retryable in most cases)
'failed': Verification failed, withdrawal cannot proceed
  (terminal state, non-retryable in most cases)
```

Withdrawals that reach the `expired` state should be retried to avoid dropping burn events.

Recommendation

Introduce logic to detect and retry expired withdrawals.

One possible approach is to persist the withdrawal IDs submitted for each burn block. At the end of `processTenure`, query the status of each withdrawal associated with the current burn block using the status endpoint.

For any withdrawal with an `expired` status, resubmit it unless it has already reached a configurable maximum number of retry attempts.

Update the latest processed block height and record it in `processedBurnBlocks` only when either:

- all withdrawals have reached terminal non-expired states, or
- all expired withdrawals have been retried the maximum allowed number of times.

If neither condition is met, skip the update so that the next `pollOnce` execution can retry the expired withdrawals.

8.3. Low Findings

[L-01] Governance Can Brick Itself

Location:

- usdcx-token.clar#L262

Description The current `usdcx-token` implementation allows the governance role to arbitrarily add and remove protocol roles.

However, there are no safeguards to prevent configurations that render the protocol unusable.

One failure case occurs when both the governance role and the minting role are completely unassigned. In this state, all core protocol functionality becomes permanently inaccessible, with the sole exception of regular transfers between users.

Recommendation

Ensure that at least one contract always retains the governance role.

For example, maintain a counter that tracks the number of contracts assigned the governance role. When removing a governance role via `set-active-protocol-caller`, assert that the resulting counter value remains greater than or equal to one. Note that in this design, enabling an already enabled role and disabling an already disabled role should be disallowed to ensure the counter does not diverge from the actual number of governance roles.

[L-02] Burn Relies on tx-sender for Authentication

Location:

- usdcx-v1.clar#L351

Description
The `usdcx-v1.burn` function assumes `tx-sender` is the token owner.

This design exposes users to phishing risks. If a user unknowingly interacts with a malicious contract, that contract can call `burn` on the user's behalf, draining their entire USDCx balance and setting the `native-recipient` to an arbitrary address.

Recommendation
Use `contract-caller` instead of `tx-sender` as the token source in `burn`.

[L-03] Burns Are Processed From Only One Minter Per Burn Block

Location:

- burn-poller.ts#L179

Description
The `usdcx-v1` contract can be upgraded at any time. When an upgrade occurs, both the old and new minter contracts may emit burn events within the same burn block.

The system also allows multiple entities to hold the minter role simultaneously, meaning more than one contract can perform burn operations in the same burn block.

However, the burn poller currently supports only a single contract identifier:

```
if (event.contract_event.contract_identifier !== usdcxV1Identifier) return;
```

As a result, burn events emitted by other valid minter contracts are ignored, while the burn block is still recorded in the database as processed.

For instance, immediately after updating the minter contract, if there is not enough delay between decommissioning the old minter and enabling the new one, pending unstable blocks may contain burn events from both contract identifiers.

In this situation, some burn events may be missed unless manual intervention is performed carefully.

Recommendation
Update the burn poller to handle minter contract changes gracefully. For example, replace the single identifier with a list of allowed contract identifiers.

[L-04] Hardcoded Zero Relay Fee Leads to Incremental Losses

Location:

- relay.ts#L103

Description The relayer currently hardcodes the fee charged for deposit minting to zero:

```
makeContractCall({
  ...usdcxV1.mint({
    depositIntent: attestation.payload,
    signature,
@>  feeAmount: 0,
  }),
  senderKey: Redacted.value(privateKey),
  network: stacksNetwork,
}),
```

As a result, each relayed deposit imposes a small transaction fee cost on the system.

Furthermore, on the source domain, deposits are not subject to any fee, and the only enforced constraint on the deposit amount is that it must be non-zero.

An attacker could therefore exploit this behavior by submitting numerous dust deposits, intentionally griefing the relayer with the transaction fees required to execute the mint calls.

Recommendation

Consider introducing a configurable minimum relay fee (e.g., `minRelayFee`) to define the lowest acceptable fee for relaying a deposit. Deposits with a `maxFee` below this configured minimum should be ignored by the relayer and instead minted manually by the remote receiver. For deposits where `maxFee` meets or exceeds `minRelayFee`, the relayer should proceed with minting and charge `feeAmount = minRelayFee`.

[L-05] Confirmation Blocks Configuration Ignored

Location:

- burn-poller.ts#L77

Description The configuration file defines `finalizeBurnBlocksConfig` as the number of required confirmation blocks:

```
export const finalizeBurnBlocksConfig = Config.number(
  "FINALIZE_BURN_BLOCKS",
).pipe(Config.withDefault(6));
```

However, the burn poller does not use this configuration value and instead relies on a hardcoded constant of 6:

```
stableBurnBlockHeight: Effect.map(Ref.get(stateRef), (s) =>
  Math.max(0, s.chainTip.burn_block_height - 6),
),
```

As a result, any value provided via configuration is ignored, rendering the setting ineffective.

Recommendation

Replace the hardcoded value with the configured `finalizeBurnBlocksConfig`.

[L-06] Mutating Token Metadata Is Problematic

Location:

- usdcx-token.clar#L216-L223

Description
The `usdcx-token` contract exposes setters that allow its name and symbol to be modified.
Many protocols in the ecosystem rely on token names and symbols to derive metadata, both on-chain and throughout their application stacks. For example, Bitflow HODLMM generates an immutable pool name at pool creation time using the token symbols:

```
(symbol (unwrap!
  (create-symbol x-token-trait y-token-trait) ERR_INVALID_POOL_SYMBOL))
(name (concat symbol "-LP"))
```

As a result, changing USDCx's metadata can cause unintended negative effects for applications that integrate with it.

Recommendation
Notify integrators in advance of any planned name or symbol changes to minimize downstream impact.

[L-07] Burn Poller Processes TenureChange Blocks

Location:

- burn-poller.ts#L103-L138
- mod.rs#L1130-L1228
- mod.rs#L1239-L1409

Description
Tenure changes on Stacks must satisfy a strict set of requirements when included in blocks to be considered valid by consensus:

- The block must contain exactly one `TransactionPayload::TenureChange(...)` transaction at index zero.
- The block must contain exactly one `TransactionPayload::Coinbase` transaction at index one, since a new tenure is being started.
- The block must not contain any other transactions.

Because these rules imply that blocks starting or extending tenures cannot contain any USDCx-related transactions, such blocks can be safely filtered out from processing.

Recommendation
Consider skipping blocks that start or extend tenures from being processed.

[L-08] Unfiltered Transaction Event Receipt Looping

Location:

- burn-poller.ts#L124-L159
- mod.rs#L898-L907

Description
The current burn poller implementation iterates over all transactions and events included in a block:

```
for (const tx of replayBlock.transactions) {
  yield * processTransaction(tx, replayBlock, burnBlock);
}
```

This is inefficient because only `ContractCall` and `SmartContract` transactions can contain xReserve burn calls. Iterating over events from other transaction types unnecessarily consumes computational resources.

Recommendation
Consider returning early in `processTransaction` when the payload type is not `TransactionPayload::ContractCall` or `TransactionPayload::SmartContract`.

[L-09] Missing Filter Causes Increasing Deposit Processing Delays

Location:

- deposit-poller.ts#L86

Description The deposit poller executes `pollOnce` on a fixed schedule. After each execution, it updates its cursor to the next page based on the link header.

Based on the xReserve testnet API behavior, the link header omits the `rel="next"` entry when no additional attestations are available. For example, the following request:

```
curl --request GET --url
// https://xreserve-api-testnet.circle.com/v1/remote-domains/10003/attestations -i
```

returns this link header:

```
<https://xreserve-api-testnet.circle.com/v1/remote-domains/10003/attestations>;
// rel="self",
<https://xreserve-api-testnet.circle.com/v1/remote-domains/10003/attestations>;
// rel="first"
```

Therefore, once the final page of the current pagination sequence is processed, the cursor is reset to an undefined value.

Whenever the cursor is reset, during the next polling round the poller will restart processing every attestation for the remote blockchain.

Over time, this can cause the number of attestations being processed to grow large enough to introduce significant delays in handling new deposit attestations.

Recommendation

The `fetchAttestations` API supports a `from` parameter that filters attestations created before a specified timestamp.

Consider introducing a configurable lookback window variable to represent the maximum age of attestations to process. Then, in the deposit poller, compute the corresponding `from` value and include it in the call to `fetchAttestations`.

8.4. QA Findings

[QA-01] Missing Event Emissions in Setter Functions

Location:

- usdcx-token.clar#L205-L212
- usdcx-token.clar#L216-L223
- usdcx-token.clar#L227-L234
- usdcx-token.clar#L262-L279

Description
The entrypoints that mutate USDCx metadata—`protocol-set-name`, `protocol-set-symbol`, `protocol-set-token-uri`, and `set-active-protocol-caller`—do not emit any events. Updating token metadata without emitting a corresponding event is generally considered an anti-pattern, as it reduces transparency and makes off-chain tracking more difficult.

Recommendation
Add `print` statements (event emissions) to each of the setters listed above to ensure metadata changes are observable off-chain.

[QA-02] Redundant Function Definitions

Location:

- usdcx-token.clar#L101
- usdcx-token.clar#L105
- usdcx-token.clar#L120

Description
The `usdcx-token` contract defines several read-only functions that are effectively redundant:

- `get-balance-available` duplicates `get-balance`
- `get-balance-locked` duplicates `get-balance`
- `is-protocol-caller` simply forwards to `validate-protocol-caller`

These functions increase the overall contract size without adding any new functionality.

Recommendation
Remove the redundant functions listed above.

[QA-03] Isolate Roles Into Constants

Location:

- usdcx-v1.clar#L156-L157
- usdcx-v1.clar#L322-L323

Description
In `usdc-token`, one-byte role flags are extracted into constants, making their purpose clear.

However, the custom `0x04` role in `usdcx-v1`, which is responsible for setting `min-withdrawal-amount`, is still used as a magic value and is not defined as a constant.

Recommendation
Consider extracting the `0x00` and `0x04` roles in `usdcx-v1` into named constants.

[QA-04] Incorrect Event Field Value

Location:

- usdcx-token.clar#L301-L302

Description The `topic` field for the `unpause` event is currently set to `"pause"`. This naming mismatch may cause confusion when reviewing printed events.

Recommendation Consider renaming the `topic` field from `"pause"` to `"unpause"`. Additionally, the events in these functions could be standardized using the following format:

```
{
  topic: "pause",
  caller: contract-caller
}
```

[QA-05] Miners Can Capture All Fees

Location:

- usdcx-v1.clar#L303

Description
The USDCx deposit flow to Stacks operates as follows:

- A user deposits USDCx on the source chain into the xReserve contract, specifying the maximum fee they are willing to pay for the deposit to be relayed.
- The xReserve attestation service produces an attestation for the deposit.

Once these steps are complete, any party can relay the deposit from the xReserve attestations API to the USDCx contract on Stacks, triggering the minting of USDCx tokens to the intended recipient. In return for performing the relay, the fulfiller may collect a fee up to the maximum amount specified by the user in step 1.

Because the current tenure miner has full control over which transactions are included in a block, as well as their ordering, they can extract value from transaction relayers through the following process:

- Wait for a deposit transaction to be submitted to the mempool.
- Create and include a deposit transaction with identical parameters but with a 0 transaction fee.
- Include the original deposit transaction after their own, thereby capturing both the USDCx relay fee and the gas fee from the relayer's transaction, which will fail but still pay for gas.

Recommendation
Since the fee mechanism is intentionally designed to incentivize deposit relays, this behavior is acceptable. We recommend formally acknowledging this issue.

[QA-06] Inconsistent Constant Naming Format

Location:

- usdcx-token.clar#L31-L35

Description The codebase consistently uses `UPPER_SNAKE_CASE` for constants and error definitions. In addition, the naming convention follows a `domain::subdomain` pattern, e.g., `ERR_UNAUTHORIZED`, `DEPOSIT_INTENT_MAGIC`.

However, the role constant definitions in `usdcx-token` conflict with this convention. They use `lowercase-kebabcase` and invert the domain/subdomain order, e.g., `governance-role`.

Recommendation Apply the following naming changes and update all code references accordingly:

```
-(define-constant governance-role 0x00)
+(define-constant ROLE_GOVERNANCE 0x00)

-(define-constant mint-role 0x01)
+(define-constant ROLE_MINT 0x01)

-(define-constant pause-role 0x02)
+(define-constant ROLE_PAUSE 0x02)
```