// Security Assessment

11.12.2025 - 11.27.2025

USDCx

Stacks Labs

HALBORN

USDCX - Stacks Labs

Prepared by: HALBORN

Last Updated 12/17/2025

Date of Engagement: November 12th, 2025 - November 27th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
9	0	0	4	4	1

TABLE OF CONTENTS

1. INTRODUCTION

Stacks engaged Halborn to conduct a security assessment on their Clarity smart contracts and attestor backend from November 12, 2025 to November 27, 2025. The security assessment scope was limited to the smart contracts and systems provided to Halborn. Commit hashes and further details can be found in the Scope section of this report.

Key risk themes included:

- Ingestion and finality gaps: at-least-once semantics are broken in the deposit poller (cursor advances past failed items), and relays return on broadcast without confirmation, enabling missed mints on mempool drop.
- Configuration/governance fragility: hardcoded destination contract/token in burn attestation creates cross-domain mismatch risk and makes rotations domain changes code-driven instead of config-governed.
- Idempotency and persistence robustness: non-idempotent burn event inserts can violate UNIQUE constraints and stall the poller during benign re-fetches or restarts.
- Input validation consistency: private key validation truncates to 32 bytes while downstream uses the full string, allowing malformed secrets to pass validation and cause key/address mismatches.

2. ASSESSMENT SUMMARY

Halborn was allocated 12 days for this engagement and assigned full-time security engineer to conduct a comprehensive review of the smart contracts and systems within scope. The engineer is an expert in blockchain and smart contract security, with advanced skills in penetration testing and smart contract exploitation, as well as extensive knowledge of multiple blockchain protocols.

The objectives of this assessment are to:

- Identify potential security vulnerabilities within the smart contracts and systems.
- Verify that the smart contract and system functionality operate as intended.

In summary, `Halborn` identified several areas for improvement to reduce the likelihood and impact of security risks, all of them were addressed by the `Stacks` `team.` The main recommendations were:

- `Ensure at-least-once semantics in the deposit poller to prevent cursor advancement past failed items.`
- `Implement confirmation checks for relays to avoid missed mints on mempool drop.`
- `Remove hardcoded destination contracts/tokens in burn attestation to mitigate cross-domain mismatch risks.`
- `Ensure idempotency in burn event inserts to prevent UNIQUE constraint violations and poller stalls.`
- `Standardize private key validation to prevent malformed secrets and key/address mismatches.`

3. TEST APPROACH AND METHODOLOGY

`Halborn` conducted a combination of manual code review and automated security testing to balance efficiency, timeliness, practicality, and accuracy within the scope of this assessment. While manual testing is crucial for identifying flaws in logic, processes, and implementation, automated testing enhances coverage of smart contracts and quickly detects deviations from established security best practices.

The following phases and associated tools were employed throughout the term of the assessment:

- Research into the platform's architecture, purpose and use.
- Manual code review and walkthrough of smart contracts and systems to identify any logical issues.
- Comprehensive assessment of the safety and usage of critical TypeScript variables and functions within scope that could lead to arithmetic-related vulnerabilities.
- Local testing using custom scripts.
- Fork testing against main networks.
- Static security analysis of scoped contracts, and imported functions.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: usdcx

(b) Assessed Commit ID: 5fe4a0d

(c) Items in scope:

- usdcx/clarity/contracts/usdcx-v1.clar
- usdcx/clarity/contracts/usdcx-token.clar
- usdcx/clarity/src/constants.ts
- usdcx/clarity/src/clarigen-types.ts
- usdcx/clarity/src/index.ts
- usdcx/clarity/src/codec.ts
- usdcx/clarity/tests/deposit-attestations.test.ts
- usdcx/clarity/tests/deposit-intent.test.ts
- usdcx/clarity/tests/usdcx-v1.test.ts
- usdcx/clarity/tests/usdcx-token.test.ts
- usdcx/clarity/tests/usdcx-proptests.test.ts
- usdcx/clarity/scripts/burn.ts
- usdcx/clarity/Clarinet.toml
- usdcx/clarity/Clarigen.toml
- usdcx/clarity/deployments/default.devnet-plan.yaml
- usdcx/clarity/deployments/default.simnet-plan.yaml
- usdcx/clarity/settings/Devnet.toml
- usdcx/clarity/vitest.config.js
- usdcx/clarity/tsconfig.json
- usdcx/clarity/tsdown.config.ts
- usdcx/clarity/package.json
- usdcx/clarity/docs/intro.md
- usdcx/clarity/docs/SUMMARY.md
- usdcx/clarity/docs/book.toml
- usdcx/clarity/deposit-intent.md
- usdcx/clarity/README.md
- usdcx/clarity/metadata/sip16.json
- usdcx/attestor/src/config.ts
- usdcx/attestor/src/deposit-poller.ts
- usdcx/attestor/src/relay.ts
- usdcx/attestor/src/burn-poller.ts
- usdcx/attestor/src/burn-intent.ts
- usdcx/attestor/src/errors.ts

- usdcx/attestor/src/types.ts
- usdcx/attestor/src/utils.ts
- usdcx/attestor/src/openapi-fetch-effect.ts
- usdcx/attestor/src/aws-kms.ts
- usdcx/attestor/src/db/schema.ts
- usdcx/attestor/src/services/circle-api-service.ts
- usdcx/attestor/src/services/stacks-api-service.ts
- usdcx/attestor/src/services/db-service.ts
- usdcx/attestor/src/services/anvil-circle-api-service.ts
- usdcx/attestor/src/services/burn-attestor-service.ts
- usdcx/attestor/src/services/burn-signer-service.ts
- usdcx/attestor/index.ts
- usdcx/attestor/vitest.config.ts
- usdcx/attestor/tsconfig.json
- usdcx/attestor/drizzle.config.ts
- usdcx/attestor/package.json
- usdcx/attestor/types/circle-openapi.d.ts
- usdcx/attestor/types/stacks-api.d.ts
- usdcx/attestor/types/xreserve-abi.json
- usdcx/attestor/types/xreserve-openapi-schema.json
- usdcx/attestor/README.md
- usdcx/bridge/src/routes/deposit.tsx
- usdcx/bridge/src/routes/accounts.tsx
- usdcx/bridge/src/routes/index.tsx
- usdcx/bridge/src/routes/api.trpc.$.tsx
- usdcx/bridge/src/routes/root.tsx
- usdcx/bridge/src/lib/stacks.ts
- usdcx/bridge/src/lib/queries.ts
- usdcx/bridge/src/lib/constants.ts
- usdcx/bridge/src/lib/eth.ts
- usdcx/bridge/src/lib/xreserve-abi.ts
- usdcx/bridge/src/lib/form-options.ts
- usdcx/bridge/src/lib/text-variants.ts
- usdcx/bridge/src/lib/utils.ts
- usdcx/bridge/src/hooks/use-accounts.ts
- usdcx/bridge/src/hooks/form-context.ts
- usdcx/bridge/src/hooks/form.ts
- usdcx/bridge/src/components/form-components.tsx
- usdcx/bridge/src/components/header.tsx
- usdcx/bridge/src/components/text.tsx
- usdcx/bridge/src/components/ui/button.tsx
- usdcx/bridge/src/components/ui/card.tsx
- usdcx/bridge/src/components/ui/input.tsx
- usdcx/bridge/src/components/ui/label.tsx

- usdcx/bridge/src/components/ui/select.tsx
- usdcx/bridge/src/components/ui/slider.tsx
- usdcx/bridge/src/components/ui/sonner.tsx
- usdcx/bridge/src/components/ui/switch.tsx
- usdcx/bridge/src/components/ui/textarea.tsx
- usdcx/bridge/src/router.tsx
- usdcx/bridge/src/routeTree.gen.ts
- usdcx/bridge/src/styles.css
- usdcx/bridge/vite.config.ts
- usdcx/bridge/tsconfig.json
- usdcx/bridge/package.json
- usdcx/bridge/components.json
- usdcx/bridge/biome.json

Out-of-Scope: Unified reserve contract on Ethereum, third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 5924798
- 2e9349c
- 6ee6ebf
- 5038ea3
- bdcf195

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	4	4	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CURSOR ADVANCES ON PARTIAL FAILURE, DROPPING UNPROCESSED ATTESTATIONS	MEDIUM	SOLVED - 11/24/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LACK OF CONFIRMATION AFTER BROADCAST MAY LEAD TO LOST MINTS ON MEMPOOL DROP	MEDIUM	RISK ACCEPTED - 12/16/2025
HARDCODED DESTINATION CONTRACT/TOKEN IN BURN ATTESTOR POSES MISCONFIGURATION AND GOVERNANCE RISK	MEDIUM	SOLVED - 12/01/2025
BURN POLLER INSERTS ARE NOT IDEMPOTENT, DUPLICATE EVENTS CAN CRASH LOOP	MEDIUM	SOLVED - 12/01/2025
MISMATCH BETWEEN CONTRACT AND CIRCLE DRAFT SPECIFICATIONS FOR MAXFEE HANDLING	LOW	SOLVED - 11/24/2025
INCONSISTENT PRIVATE KEY VALIDATION ALLOWS MALFORMED SECRETS	LOW	SOLVED - 11/21/2025
REMOTE TOKEN CONSTRUCTION ASSUMES 28-BYTE CONSENSUS ENCODING	LOW	RISK ACCEPTED - 12/09/2025
INCOMPLETE HOOK DATA LENGTH HANDLING MAY CAUSE PANIC WITH FUTURE CHANGES	LOW	RISK ACCEPTED - 12/09/2025
MISSING UNIQUE REQUESTID IN CIRCLE WITHDRAW LEADING TO IDEMPOTENCY AND ANTI-REPLAY RISK	INFORMATIONAL	SOLVED - 12/01/2025

7. FINDINGS & TECH DETAILS

7.1 CURSOR ADVANCES ON PARTIAL FAILURE, DROPPING UNPROCESSED ATTESTATIONS

// MEDIUM

Description

The `deposit` poller retrieves attestations from the Circle API page by page and attempts to relay each one to Stacks. Individual attestations are retried with an exponential backoff up to `maxRetriesConfig`; if they still fail, the error is logged and processing continues with the next item.

After finishing a page, the poller advances the pagination cursor unconditionally to `response.nextPageCursor` and increments the processed counter by the page size, regardless of whether some items ultimately failed after their per-item retries.

This control flow breaks at-least-once ingestion semantics: any attestation that exceeded its retry budget will not be retried on subsequent iterations because the cursor has moved past it. As a result, deposits can become stuck and minting on Stacks can be delayed until a manual backfill or replay is performed.

The issue is most visible under transient relay instability (RPC outages, congestion, gas issues) or repeated application-level rejections, where gaps appear in the ingestion stream, operational overhead increases to diagnose and recover missed items, and user-facing settlement times are prolonged.

The problem affects any environment configured through `remoteDomainConfig` whenever one or more attestations in a fetched page fail conclusively within the retry window.

Code Location

The cursor is advanced unconditionally in the following code:

```
yield* Ref.update(stateRef, (s) => ({
  cursor: response.nextPageCursor,
  processedCount: s.processedCount + attestationCount,
}));
```

Proof of Concept

The following proof of concept highlights the failure on the cursor advancement:

```
// attestor/tests/poller.cursor-advance-bug.test.ts
import { describe, it, expect } from "vitest";
import { Effect, Layer, TestClock, Ref, Fiber, TestContext } from "effect";
import { pollAttestations } from "../src/poller.js";
import { RelayService } from "../src/relay.js";
import {
```

```
  generateMockAttestations,
  makeMockCircleAttestationService,
} from "./mocks/circle-api-mock.js";
import { CircleApiService } from "../src/services/circle-api-service.js";

describe("cursor processing semantics", () => {
  it("retries/acknowledges failed attestations before advancing cursor (desired behavior)", async () =>
    const [A, B, C] = generateMockAttestations(3);
    const { service: circleService } = makeMockCircleAttestationService([A, B, C]);

    const processedRef = Ref.unsafeMake<string[]>([]);
    const relay = RelayService.of({
      relayAttestation: (att) =>
        Effect.gen(function* () {
          // Permanently fail A; succeed others
          if (Buffer.from(att.messageHash).toString("hex") === Buffer.from(A.messageHash).toString("hex"
            return yield* Effect.fail(new Error("perm-fail"));
          }
          yield* Ref.update(processedRef, (xs) => [
            ...xs,
            Buffer.from(att.messageHash).toString("hex"),
          ]);
          return { txId: "0xok", messageHash: att.messageHash };
        }),
    });

    const program = Effect.gen(function* () {
      const fiber = yield* Effect.fork(
        pollAttestations().pipe(
          Effect.provide(Layer.succeed(CircleApiService, circleService)),
          Effect.provide(Layer.succeed(RelayService, relay)),
        ),
      );

      // First iteration: processes page [A,B]. B succeeds; A fails after retries (current behavior adva
      yield* TestClock.adjust("6 seconds");
      // Second iteration: processes C on next page
      yield* TestClock.adjust("5 seconds");

      const processed = yield* Ref.get(processedRef);
      const hexA = Buffer.from(A.messageHash).toString("hex");
      const hexB = Buffer.from(B.messageHash).toString("hex");
      const hexC = Buffer.from(C.messageHash).toString("hex");

      expect(processed).toContain(hexB);
      expect(processed).toContain(hexC);
      // Pre-patch this will FAIL because A is dropped; after fix, A should be retried/processed
      expect(processed).toContain(hexA);

      yield* Fiber.interrupt(fiber);
    });

    await Effect.runPromise(
      program.pipe(Effect.provide(TestContext.TestContext)),
    );
  });
});
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

To address this issue, it is recommended to advance the cursor only after all items in the page have been acknowledged. Alternatively, persist per-message-hash acknowledgments (processed, failed, dead-letter) and retry failed items independently of the page cursor to avoid dropping unprocessed attestations.

Remediation Comment

SOLVED: The `Stacks` team solved this finding by advancing the cursor only after all items on the page have been acknowledged.

Remediation Hash

https://github.com/stx-labs/usdcx/commit/59247984b2bae3c934220c3bd91dee14c5098f61

7.2 LACK OF CONFIRMATION AFTER BROADCAST MAY LEAD TO LOST MINTS ON MEMPOOL DROP

// MEDIUM

Description

The vulnerability occurs when the relay broadcasts a mint contract call transaction and considers it successful if the broadcast API accepts it, without waiting for chain confirmation or verifying the final execution status. The deposit poller interprets this as a successful relay and proceeds with the next attestation, advancing the pagination cursor in subsequent iterations. If the transaction is later dropped from the mempool (due to reasons such as insufficient fees, node restarts, or RBF/nonce conflicts) or if it fails upon execution at the contract level, the system does not detect the failure and does not re-enqueue the deposit for another attempt.

This results in a gap where a deposit can be marked as relayed but never minted on-chain, necessitating manual replay or backfill for recovery. This issue can potentially lead to the loss or indefinite delay of intended mints under transient network or node conditions, and increases the operational burden to discover and reconcile missed deposits, as there is no confirmation, receipt inspection, or retry policy at the confirmation stage.

Code Location

Broadcast considered sufficient; returns without confirmation:

```
Failed to broadcast transaction: ${receipt.reason}: ${receipt.error}
```

, cause: `${receipt.reason}: ${receipt.error}` , }),); } yield* Effect.logInfo("Relayed attestation to Stacks", { txId: tx.txid(), messageHash: hex.encode(attestation.messageHash), }); return { txId: tx.txid(), messageHash: attestation.messageHash, };" language="typescript" linenumbers="true" wordwrap="false" tabsize="2" shouldfocus="false" highlightedlines="" datastart="" class="language-typescript">`const receipt = yield* Effect.tryPromise({ try: () => broadcastTransaction({ transaction: tx, network, }), catch: (cause) => new RelayError({ message: "Failed to broadcast transaction", cause, }), }); if ("error" in receipt) { return yield* Effect.fail(new RelayError({ message: Failed to broadcast transaction: ${receipt.reason}: ${receipt.error} , cause: ${receipt.reason}: ${receipt.error} , }),); } yield* Effect.logInfo("Relayed attestation to Stacks", { txId: tx.txid(), messageHash: hex.encode(attestation.messageHash), }); return { txId: tx.txid(), messageHash: attestation.messageHash, };`

The deposit poller advances once `relayAttestation` resolves, with no confirmation step:

```
Processing attestation: ${attestation.messageHash}
```

,); const result = yield* relayService.relayAttestation(attestation); yield* Effect.logDebug(`Relayed attestation ${hex.encode(attestation.messageHash)} -> Stacks TX: ${result.txId}` ,); }); }" language="typescript" linenumbers="true" wordwrap="false" tabsize="2" shouldfocus="false" highlightedlines="" datastart="" class="language-typescript">`for (const attestation of`

```
response.attestations) { yield* Effect.gen(function* () { yield* Effect.logDebug(
  Processing attestation: ${attestation.messageHash}  , ); const result = yield*
relayService.relayAttestation(attestation); yield* Effect.logDebug(
  Relayed attestation ${hex.encode(attestation.messageHash)} -> Stacks TX: ${result.txId}  , ); }); }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

To address this issue, it is recommended to add a confirmation step in the relay flow that polls transaction status via the Stacks API until success or a bounded timeout. The cursor should only advance on confirmed success. On timeout or failure, the system should re-enqueue the transaction with a fee-bump (replacement) policy and alert after a configured number of attempts. Additionally, maintain a pending set keyed by transaction ID or message hash that is cleared only upon confirmed success.

Remediation Comment

RISK ACCPTED: The `Stacks` team has accepted the risk of this finding with the following statement:

> There are many scenarios where the mint transaction might fail, such as: - The Circle attestation key that signed the deposit is invalid (perhaps it was revoked) - The recipient is invalid (users can input anything on the ETH side) - The `maxFee` or `amount` is greater than u128::max (theoretically possible, not practically possible with only 6 decimals) Thus, I think the remediation should be "the transaction was confirmed, even if it failed", which is similar but different than the proposed remediation. We accept that this is not implemented at this time, and we intend to implement this functionality shortly.

7.3 HARDCODED DESTINATION CONTRACT/TOKEN IN BURN ATTESTOR POSES MISCONFIGURATION AND GOVERNANCE RISK

// MEDIUM

Description

The `burn attestor` constructs the `BurnIntent` with a dynamic `destinationDomain` sourced from the burn event but uses hardcoded `destinationContract` and `destinationToken` values that do not vary with the selected destination.

Circle's cross-domain specification requires these fields to match the destination domain's canonical contract and token addresses. Treating them as static works for a single-domain prototype but becomes fragile in multi-domain or production use: an operator switching the target domain or onboarding an additional domain would silently produce a mismatch between domain and addresses, leading Circle to reject withdrawals at validation time.

Because these parameters are embedded in code rather than externalized configuration, any rotation, migration, or domain expansion requires a code change and redeploy instead of a controlled config rollout, increasing governance and operational risk.

The practical effect is withdrawal unavailability (requests are rejected before funds leave custody), plus a higher chance of misconfiguration persisting undetected until users experience failed withdrawals.

Code Location

The hardcoded values for `destinationContract` and `destinationToken` are shown below:

```
destinationContract: "0x00000000000000000000000002222222d7164433c4c09b0b0d809a9b52c04c205",
destinationToken: "0x000000000000000000000000a0b86991c6218b36c1d19d4a2e9eb0ce3606eb48",
destinationCaller: "0x0000000000000000000000000000000000000000000000000000000000000000",
```

The `destinationDomain` is dynamic (`burnEventData.nativeDomain`), but the corresponding contract/token remain static, creating a mismatch risk across domains.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

To address this issue, it is recommended to externalize the destination contract, token, and caller into configuration keyed by `destinationDomain` per environment; validate the configuration at startup by fetching Circle metadata (`/v1/info`) and fail fast on any mismatch; enforce guards to reject operations for domains without configured mappings before submitting to Circle, and add automated tests (property tests over domain tuples and unit tests for guard behavior) to ensure ongoing correctness.

Remediation Comment

SOLVED: The `Stacks` team solved this finding by externalizing the destination contract, token, and caller into configuration.

Remediation Hash

https://github.com/stx-labs/usdcx/commit/2e9349c0582c2b116061db0e42723f41ea2f0e0a

7.4 BURN POLLER INSERTS ARE NOT IDEMPOTENT, DUPLICATE EVENTS CAN CRASH LOOP

// MEDIUM

Description

The `burn_poller` replays Stacks blocks and records every discovered burn event into the `burn_events` table. The schema enforces a uniqueness constraint on the `(txId, eventIndex)` pair to prevent duplicates. However, the insertion path issues an unconditional insert without specifying an upsert or ignore policy on conflict.

This can lead to issues when the same burn event is observed more than once, such as after a service restart before the corresponding checkpoint (`processed_burn_blocks`) is persisted, due to a benign re-fetch of the same tenure, or from an upstream API quirk. The insert then violates the unique index and throws an exception. Since the exception is not handled at the insert boundary, it propagates through the per-event processing and into the polling iteration's retry mechanism, causing repeated retries, noisy logs, and temporary loss of liveness for the poller until the duplicate condition subsides or the retry window elapses.

This failure mode is particularly likely around restarts or partial progress windows where events have been recorded but the tenure checkpoint has not yet been written, causing the same block range to be reprocessed and duplicate inserts to be attempted.

Code Location

The unique constraint is defined in the schema:

```
(table) => [ unique("tx_id_event_index_unique").on(table.txId, table.eventIndex), ], );
```

The insert function does not handle conflicts:

```
export const insertBurnEvent = (event: typeof burnEvents.$inferInsert) => Effect.gen(function* () { cons
```

The function call that triggers the insert:

```
yield* insertBurnEvent({ txId: strip0x(tx.txid), eventIndex: event.event_index, sender: burnData.sender,
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

To address this issue, it is recommended to make burn-event insertion idempotent by using an upsert keyed by transaction ID and event index, treating duplicates as benign and continuing processing; optionally update observability timestamps on conflict and, if multi-write sequences are introduced, wrap them in a transaction.

Remediation Comment

SOLVED: The `Stacks` team solved this finding by making burn poller inserts idempotent, adding onConflictDoNothing on burnEvents and relayedDeposits and refactoring DB calls through a withDb helper, so duplicate events no longer crash the loop.

Remediation Hash

https://github.com/stx-labs/usdcx/commit/6ee6ebf09cd09ecdd394d5eebd8154d4a44a6658

7.5 MISMATCH BETWEEN CONTRACT AND CIRCLE DRAFT SPECIFICATIONS FOR MAXFEE HANDLING

// LOW

Description

The on-chain validation in `parse-and-validate-deposit-intent` requires that the intent amount strictly exceeds the declared `max-fee`, implemented as `asserts!(> amount (get max-fee parsed-intent)) ERR_INVALID_DEPOSIT_MAX_FEE_GTE_AMOUNT`. This strict inequality rejects deposit intents where the amount equals `maxFee`, even when the actual `feeAmount` is strictly less than `maxFee` and the resulting `mintAmount` would be positive.

In contrast, the Circle draft specification permits `depositIntent.amount` to be greater than or equal to `depositIntent.maxFee`, with a separate guarantee that `depositIntent.maxFee` is greater than or equal to `feeAmount`.

The divergence creates a compatibility gap between the contract and integrator expectations: valid intents under the spec `(amount == maxFee and feeAmount < maxFee)` will revert on-chain with `ERR_INVALID_DEPOSIT_MAX_FEE_GTE_AMOUNT`, unnecessarily blocking deposits and minting and forcing relayers or users to resubmit intents that are otherwise compliant.

Concretely, a Circle-compliant relayer may cap fees conservatively with `maxFee` equal to `amount` while charging a smaller `feeAmount`; the current contract logic rejects such intents even though the fee cap is respected, the charged fee is below the cap, and the computed mint would remain non-negative. Aligning the check to allow `amount >= maxFee` (while preserving the independent constraint `maxFee >= feeAmount`) restores compatibility with the spec and prevents avoidable reverts for otherwise valid deposits.

Code Location

The current validation logic in the contract:

```
asserts!(> amount (get max-fee parsed-intent)) ERR_INVALID_DEPOSIT_MAX_FEE_GTE_AMOUNT
```

Proof of Concept

The following proof of concept shows that the contract currently rejects `amount == maxFee`:

```
// clarity/tests/usdcx-v1.amount-eq-maxfee.spec.test.ts
import { describe, it, expect } from "vitest";
import { txErr, txOk } from "@clarigen/test";
import {
  usdcxV1,
  errors,
  getPublicKey,
  randomSecretKey,
```

```
  buildIntentBytes,
  signIntent,
  deployer,
} from "./helpers";

describe("amount == maxFee spec behavior", () => {
  it("accepts amount == maxFee when feeAmount < maxFee (per Circle spec)", () => {
    const sk = randomSecretKey();
    const pk = getPublicKey(sk);
    // Add attestor
    txOk(usdcxV1.addOrRemoveCircleAttestor(pk, true), deployer);

    const amount = 100_000n;
    const maxFee = amount; // equal, allowed by Circle draft
    const feeAmount = maxFee - 1n; // relayer fee strictly less than cap

    const intent = buildIntentBytes({ amount, maxFee });
    const sig = signIntent(intent, sk);

    // Pre-patch this should FAIL (contract currently rejects amount == maxFee)
    txOk(usdcxV1.mint(intent, sig, feeAmount), deployer);
  });
});
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

To address this issue, it is recommended to align the contract with the Circle draft by treating transfers where the amount equals the declared maximum fee as valid, while preserving runtime validations that prevent the fee from exceeding the declared maximum and ensure the resulting minted amount remains strictly positive.

Remediation Comment

SOLVED: The `Stacks` team solved this finding by aligning the contract with Circle's draft specification.

Remediation Hash

https://github.com/stx-labs/usdcx/commit/5038ea316c2c7b51853d5dd0d062202450887a86

7.6 INCONSISTENT PRIVATE KEY VALIDATION ALLOWS MALFORMED SECRETS

// LOW

Description

The private key validator decodes the provided hex string and checks `isValidSecretKey` against only the first 32 bytes of the decoded data. This means an over-long hex string can pass validation as long as its first 32 bytes form a valid secret, even if the remaining bytes are extraneous. Subsequent code paths do not reuse the truncated bytes; instead, they pass the original, untrimmed string to consumers such as `getAddressFromPrivateKey`.

This inconsistency allows subtly malformed secrets to be accepted during configuration while producing key/address material derived from the full string later, increasing the likelihood of silent key/address mismatches and hard-to-diagnose misconfiguration.

While this does not directly create a funds-at-risk condition, it degrades reliability and operational safety by letting invalid inputs slip through validation and only surface as downstream signing or address inconsistencies.

Code Location

The private key validation logic is implemented as follows:

```
const privateKeySchema = Schema.String.pipe( Schema.filter((s) => { return ( secp256k1.utils.isValidSecr
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

To address this issue, it is recommended to enforce strict private key validation by requiring an exact 32-byte hexadecimal value and rejecting any non-hex or incorrectly sized input. Normalize the input to a canonical format (lowercase hex without the 0x prefix) and ensure that only the normalized form is used throughout the system.

Remediation Comment

SOLVED: The `Stacks` team solved this finding by requiring an exact 32-byte hexadecimal value to enforce strict private key validation.

Remediation Hash

https://github.com/stx-labs/usdcx/commit/bdcf195a57a8cfd470b34cbf24dfb049dfe2280d

7.7 REMOTE TOKEN CONSTRUCTION ASSUMES 28-BYTE CONSENSUS ENCODING

// LOW

Description

The contract derives the expected 32-byte remote token by left-padding four zero bytes to the consensus serialization of the `.usdcx` contract principal and enforces a fixed 28-byte size via `as-max-len? … u28` combined with `unwrap-panic`.

This approach is correct with today's consensus encoding but implicitly bakes in a strict length invariant. If a future network version or serialization change alters the principal's consensus length, the function would trap rather than return a typed error, converting what should be a clean validation failure into a transaction panic.

In that scenario, deposit intents depending on remote-token equality would fail abruptly, reducing availability and making troubleshooting harder. The fixed-length assumption also weakens resilience to format evolution; the invariant is not surfaced defensively (e.g., as a typed error) and is not explicitly covered by tests, increasing maintenance risk as the ecosystem evolves.

This is correct for current networks and the present encoding. However, the use of `as-max-len? … u28` + `unwrap-panic` bakes in a fixed length. If the consensus encoding length changes (future network/version differences), the function could trap instead of returning a typed error, causing availability issues for deposits.

Code Location

The function that derives the remote token:

```
(define-read-only (get-valid-remote-token)
  (concat 0x00000000
    (unwrap-panic (as-max-len? (unwrap-panic (to-consensus-buff? .usdcx)) u28))
  )
)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

To address this issue, it is recommended to remove the fixed 28-byte assumption by deriving the consensus-encoded address at runtime, returning an explicit error if derivation fails, ensuring the encoded

address does not exceed 32 bytes and left-padding with zeros to exactly 32 bytes, and adding small per-network tests to validate the 32-byte result to avoid brittle failures if encoding changes.

Remediation Comment

RISK ACCEPTED: The `Stacks` team has accepted this risk, citing the expectation that consensus rules will remain unchanged in the following statement:

> We are confident that the consensus rules for encoding Stacks principals will not change in the future.

7.8 INCOMPLETE HOOK DATA LENGTH HANDLING MAY CAUSE PANIC WITH FUTURE CHANGES

// LOW

Description

The contract derives `hook-data-len` from bytes 236–240 and asserts that the overall buffer length equals 240 + `hook-data-len`. It then slices and constrains the hook-data segment with `as-max-len?` and `unwrap-panic`. Currently, the deposit intent is statically typed as a 320-byte buffer, effectively capping `hook-data-len` at 80. This ensures the extraction cannot exceed `u80` and does not trigger a panic in practice. However, this safety is implicit and relies on the coupling between the static buffer size and the equality assertion.

If a future version increases the maximum deposit intent size or the equality check is refactored or removed, malformed inputs could cause `unwrap-panic` to trigger during hook-data extraction, converting a validation error into a transaction trap. Such traps reduce availability (localized DoS on deposit relays) and hinder diagnostics by failing abruptly rather than returning a typed error. The invariant “hook-data length ≤ 80” is not explicit at the extraction site, raising maintainability risks and making future format evolution more error-prone.

Code Location

The following code illustrates the current handling of hook-data length:

```
(unwrap-panic (as-max-len?
  (unwrap-panic (slice? deposit-intent u240 (+ u240 hook-data-len)))
  u80
))
```

Currently, because `deposit-intent` is typed as `(buff 320)`, the equality `len == 240 + hook-data-len` implicitly constrains `hook-data-len ≤ 80`, so extraction cannot exceed `u80` and does not panic in practice.

However, this safety depends on an implicit coupling between (a) the static max length `(buff 320)` and (b) the prior equality check. If either changes in the future (the intent buffer max grows for a new version, or the equality check is refactored), the `unwrap-panic` chain may begin trapping on malformed inputs instead of returning a typed `err`. This turns a validation failure into a transaction trap (availability impact) and makes diagnostics harder.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

To address this issue, enforce an explicit maximum hook-data length before any slicing (rejecting values above 80 bytes). Replace panics with typed error returns for malformed inputs, and add boundary tests ensuring lengths 0, 1, and 80 succeed while 81 and any mismatch between declared and actual lengths are rejected.

Remediation Comment

RISK ACCEPTED: The `Stacks` team accepted this risk, citing the expectation that consensus rules will remain unchanged so existing checks are sufficient.

7.9 MISSING UNIQUE REQUESTID IN CIRCLE WITHDRAW LEADING TO IDEMPOTENCY AND ANTI-REPLAY RISK

// INFORMATIONAL

Description

The `withdraw` (burn intent) request to Circle currently sets the `requestId` as `"TODO"` instead of a unique UUID. The Circle draft specification requires a UUID to ensure idempotency and prevent replay or duplicate processing. This issue is acknowledged in the code as a placeholder, indicating it is intended for future hardening rather than a current security risk.

Code Location

The code responsible for setting the `requestId` is shown below:

```
const { data, error, response } = yield* Effect.tryPromise({
  try: () =>
    client.POST("/v1/withdraw", {
      body: {
        requestId: "TODO",
        batches: [
          {
            burnIntents,
            burnSignatures,
            burnTxId,
            requireCircleRelay,
          },
        ],
      },
    }),
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

To address this issue, replace the placeholder `requestId: "TODO"` with a unique UUID for each request. This can be achieved by generating a UUID using a library such as `uuid` in JavaScript, ensuring each `withdraw` request is uniquely identifiable and idempotent.

Remediation Comment

SOLVED: The `Stacks` team solved this finding by generating a UUID and remove the previous "TODO".

Remediation Hash

https://github.com/stx-labs/usdcx/commit/2e9349c0582c2b116061db0e42723f41ea2f0e0a

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.