# CSOV Bridge Security Audit

## Info

- Auditor: MEVNOMICS
- Commit hash:
  - csov-bridge: 1aa634b
  - orion-vesting: 74b42da
- Repository:
  - csov-bridge
  - orion-vesting
- Documentation quality: Medium

**Disclaimer** This audit covers the CSOV Bridge system including both the `csov-bridge` program (cross-chain bridge from Polkadot to Solana) and the `orion-vesting` program (time-locked token vesting). The audit focuses on cryptographic verification mechanisms, cross-program invocations, access control, PDA validation, signature verification, and economic attack vectors. The audit does not cover the underlying Polkadot blockchain state or the external Schnorrkel signature library implementation. Critical issues related to signature verification economics, fund custody, and vesting schedule integrity have been identified and should be addressed before production deployment.

**Fixes Update:** All critical and high-severity vulnerabilities identified in this audit have been successfully fixed by the development team. All 5 High severity issues and all 6 Medium severity issues have been resolved with proper bounds checking, validation limits, and architectural improvements.

## Key Actors and their Capabilities

1. **Bridge Users (Clients)**: Can initiate bridge requests by calling SRW init, providing Polkadot source address and signed Solana target address. Users must pay 0.02 SOL bridge fee per request. Users receive their signature checked incrementally over multiple transactions to stay within Solana compute limits. Users' bridged tokens are automatically locked in a 48-month vesting schedule with 12-month cliff. Users can claim vested tokens anytime after the cliff period via the vesting contract. Users are subject to the drop-dead date (Jan 1, 2027) - all bridge operations cease permanently after this date.

2. **Worker/Backend Operator**: Receives bridge fees (0.02 SOL per request) from users. Submits Polkadot block data to Solana for verification. Creates Block Progress Struct (BPS) PDAs with block headers, extrinsics, and Merkle proofs. Triggers seal verification and finalization transactions. Must provide correct Aura authority signatures and valid Merkle proofs. Can initiate signature verification checks but cannot manipulate the cryptographic verification results.

3. **Pauser (Bridge Contract)**: Dedicated address with authority to pause/unpause the bridge contract. Pause blocks all user operations: SRW init, signature checks, data submission, and finalization. Pause does NOT block unpause instruction itself (instruction 1). Pause does NOT block initialization (instruction 0). Separate role allows emergency response without full deployment privileges. The pauser address is set during initialization and stored in the pausing PDA.

4. **Pauser (Vesting Contract)**: Separate pauser for the vesting contract with authority to pause/unpause vesting operations. Pause blocks: adding new vesting schedules and releasing vested tokens. Pause does NOT block unpause instruction. Pause does NOT block initialization. Pauser address set during vesting contract initialization.

5. **Approved Vester Accounts**: Two hardcoded addresses authorized to add new vesting schedules to the vesting contract: `APPROVED_DOT_SOL_BRIDGE_VESTER` (61kggoHzzoB8dsAVVXFGcyT5Wkc9QdvMh2ufdkV) and `APPROVED_WCSOV_BRIDGE_VESTER` (DKnFMzvu62vSKX9cfgpja9FkikUYf93XhQhFeSgAndGd). Only these accounts can call the `new()` instruction (10) to add vesting entries. The bridge's vester PDA must match one of these addresses to successfully invoke the vesting contract during finalization.

6. **Aura Authorities**: Polkadot validator nodes authorized to seal blocks. Currently hardcoded as single authority: `d43593c715fdd31c61141abd04a99fd6822c8558854ccde39a5684e7a56da27d` . Block seal signatures must come from this authority for BPS verification to succeed. No mechanism exists to update or rotate authority list on Solana side.

7. **Mint Authority PDA**: Program-derived account with authority to mint tokens from the hardcoded token mint ( `9hx6TxrPPxA2bkYM61F1HBcrZdUxaVnpVcxV3nuoNCDs` ). Derived using seed `b"minter"` in the vesting program. Signs `mint_to` instructions to release vested tokens to users. No validation exists to confirm this PDA actually has mint authority during initialization.

## Operational Considerations

1. **Incremental SR25519 Signature Verification**: The bridge uses a custom Schnorrkel implementation with stateful, incremental verification to handle Solana's 200k compute unit limit per transaction. Signature verification is split across multiple transactions with progress stored in `DotSigCheckStruct.i` (iteration counter) and `projective_point` (15 u64 values = 120 bytes). Initial state has `i = 300` and empty projective point. Each call to `verify()` advances the computation and updates the state via callback. Final states are `i = 1000` (signature valid) or `i = 1001` (signature invalid). For SRW PDAs, this verifies the Polkadot address owner signed the Solana target address. For BPS PDAs, this verifies the Aura authority signed the block pre-header hash.

2. **Bridge Request Flow with SRW PDAs**: Users call `srw::init()` (instruction 5) which: (1) Checks if SRW PDA already exists for the Polkadot source address, (2) If exists and Open with valid signature (i=1000), rejects with `ERR_PDA_SRW_INVOLVED_IN_ACTIVE_BRIDGING_OPERATION`, (3) Transfers 0.02 SOL bridge fee from client to worker, (4) Parses DOT source address (32 bytes), tagged Solana target address (base58 with XML-style tags), and signature (64 bytes), (5) Creates or overwrites SRW PDA seeded by DOT source address, (6) Stores signature verification state with status=Open and i=300. Users then repeatedly call `srw::check()` (instruction 6) to advance signature verification until i=1000 or i=1001.

3. **Block Verification and Finalization Flow**: Backend submits Polkadot block data via: (1) `data::init_data_pda()` (instruction 10) creates data collector PDA and stores first chunk, (2) `data::continue_data_pda()` (instruction 11) appends additional chunks if block data exceeds transaction limits, (3) `bps::init()` (instruction 20) reads data PDA and creates BlockProgressStruct with block number, extrinsics root, seal, signing key, lock operations list, and Merkle proof for events, (4) Backend validates extrinsics root by computing trie hash and comparing to block header, (5) Backend verifies events via Merkle proof against state root to prove lock operations succeeded, (6) Seal verification via `bps::seal_check()` (instruction 30) verifies Aura authority signature on block pre-header, (7) `finalize::finalize()` (instruction 40) matches proven locks to valid SRW PDAs and invokes vesting contract.

4. **Cross-Program Invocation to Vesting Contract**: The `finalize()` function is the only point where the bridge invokes the vesting contract. It: (1) Verifies BPS status is SealOk, (2) Iterates through all lock operations in BPS, (3) For each lock, finds corresponding SRW PDA by DOT source address, (4) Checks SRW exists, is Open, and has valid signature (i=1000), (5) Marks SRW as Closed and lock as vested in BPS, (6) Collects (source, target, amount) tuples, (7) Divides amount by 1000 (DECIMALS_DIVISOR) to convert from 12-decimal CSOV to 9-decimal ORION, (8) Invokes vesting contract's `new()` instruction with vester PDA signature, (9) If all locks in BPS are vested, marks BPS as Done. The vester PDA must be hardcoded as `APPROVED_DOT_SOL_BRIDGE_VESTER` for CPI to

succeed.

5. **Vesting Schedule Accounting**: Each vesting entry stores: source (32 bytes DOT address), target (32 bytes Solana address), total_amount (u128), free (u128 - starts at 0), ts (i64 timestamp when created). The vesting list PDA is dynamically sized and can be reallocated as entries are added. Release calculation: (1) Check `months_passed = (current_time - ts) / 2,629,800`, (2) If months_passed ≤ 12 (cliff), return (no tokens released), (3) If months_passed == 48, set free = total_amount, (4) Else: `months_over_cliff = months_passed - 12`, `amt_per_month = total_amount / 36`, `should_be_free = min(months_over_cliff * amt_per_month, total_amount)`, (5) Mint `should_be_free - free` tokens. The calculation uses integer division which causes small rounding errors in favor of users (underpayment risk is negligible).

6. **PDA Derivation Security**: The bridge uses multiple PDA types: (1) Vester PDA: seed `b"vester"`, must match approved vester address, (2) Pausing PDA: seed `b"pausing"`, stores pauser address and pause state, (3) SRW PDA: seed = DOT source address (32 bytes variable), stores signature verification state, (4) Data Collector PDA: seed provided in instruction data (variable length), stores large block data, (5) BPS PDA: seed provided in instruction data (typically block number), stores block verification state. All PDA validations use `Pubkey::find_program_address()` to recompute and compare with provided account. However, seed validation is inconsistent - some functions trust instruction data without additional validation.

7. **Decimal Conversion Risk**: CSOV token (Polkadot side) has 12 decimals while ORION token (Solana side) has 9 decimals. The `finalize()` function divides lock amounts by 1000: `bps.locks[i].amount / DECIMALS_DIVISOR`. This truncates any fractional amounts less than 0.001 CSOV (1e9 base units). For example, a lock of 1.2345 CSOV (1,234,500,000,000 base units) becomes 1.234 ORION (1,234,500,000 base units) - losing 500,000 base units. Over many small transactions, this dust accumulates in the system with no recovery mechanism.

8. **Drop-Dead Date Enforcement**: The main entry point checks `Clock::get()?.unix_timestamp >= DROP_DEAD_TIMESTAMP` (1,798,761,600 = Jan 1 2027 00:00:00 UTC) and returns `PASSED_DROP_DEAD_DATE` error for all instructions. EXCEPT instruction 0 (init) and instruction 1 (unpause) which execute BEFORE the timestamp check. This means after the drop-dead date, the contract can still be unpaused and initialized (if not already initialized), but no other operations work. Users' funds that are already in vesting schedules remain accessible forever since the vesting contract has no drop-dead date.

9. **Signature Verification Economic Model**: Each signature verification requires multiple transactions to complete due to compute limits. The current implementation stores progress in PDA state, allowing anyone to continue verification. Gas costs for verification are paid by whoever calls the `check()` function - typically the backend operator or user. An adversarial user could create an SRW with an invalid signature, forcing the operator to spend gas to verify it fails (reaches i=1001). The 0.02 SOL fee is non-refundable even if signature verification fails. An attacker spending 0.02 SOL per invalid request could force operators to waste gas on verification.

10. **Pausability Bypass via Instruction Ordering**: The pause check happens AFTER instruction dispatch for init (0) and unpause (1). The main process_instruction reads instruction byte, checks if it's 0 or 1, and executes immediately. For all other instructions, it first gets the pausing PDA and calls `assure_unpaused()`. This means: (1) Init can be called when paused (but only if not already initialized), (2) Unpause can be called when paused (by design, to unpause), (3) All other instructions (2, 5, 6, 10, 11, 20, 30, 40) respect pause state. The same pattern exists in the vesting contract.

11. **Data Collector PDA Lifecycle**: Data PDAs are created per block to store large block data that doesn't fit in a single transaction. The lifecycle is: (1) `init_data_pda()` creates PDA with computed size and stores first chunk, (2) `continue_data_pda()` can be called multiple times to append data, (3) `bps::init()` reads the complete data and processes it, (4) Data PDA remains allocated after use with no cleanup mechanism. This causes permanent storage bloat as each block requires ~1-2 KB data PDA that's never closed. Additionally, there's no verification that data collection is complete before `bps::init()` reads it - backend could accidentally read incomplete data.

12. **Hardcoded Configuration Risks**: Multiple critical addresses are hardcoded: (1) Vesting contract: `HQ7Aph4ECQeDwiUTgu2DudN6Df1yeePkJGnj5iCMM5My`, (2) Approved vesters: 2 addresses, (3) Token mint: `9hx6TxrPPxA2bkYM61F1HBcrZdUxaVnpVcxV3nuoNCDs`, (4) Aura authority: 1 public key. If any of these need to change (upgrade, key rotation, migration), the programs must be redeployed with updated values. There are no on-chain mechanisms to update these addresses, and no validation that they implement expected interfaces or have expected permissions during deployment.

# Issues

## [High] Integer Underflow in contains() Function Causes Program Panic

**Status:Fixed**

**Description** The `contains()` utility function has an integer underflow bug when `big.len() < lil.len()`. Line 167 performs `big.len() - lil_len` without checking if `big.len() >= lil_len` first, causing subtraction with overflow panic in debug mode or integer wrapping in release mode. This function is used in critical merkle proof validation code at line 209 of `bps.rs` and line 20 of `merkle.rs`.

**Exploit Scenario**

1. Attacker calls `init_data_pda()` (no authorization beyond "contract not paused")
2. Attacker crafts malicious block data where `first_merkle_line` is very short (e.g., 5 bytes)
3. Success events are typically 20+ bytes in length
4. Attacker calls `bps::init()` which processes the malicious data
5. At line 209, code executes `contains(first_merkle_line, success)` where first_merkle_line (5 bytes) < success (20 bytes)
6. Program panics: "attempt to subtract with overflow"
7. All subsequent attempts to process this block fail with panic
8. Bridge is effectively shut down for this block seed
9. If attacker uses legitimate block seeds, they can permanently block real bridge operations
10. Attacker can repeat for multiple block numbers, causing widespread DoS

**Files**

- csov-bridge/src/utils.rs (lines 164-182)
- csov-bridge/src/bps.rs (line 209 - event verification)
- csov-bridge/src/merkle.rs (line 20 - merkle proof validation)

**Recommendation** Fix the contains function to handle edge cases:

```
pub fn contains(big: [u8], lil: [u8]) -> bool {

    if lil.is_empty() {

        return true;

    }

    if big.len() < lil.len() {

        return false; // Fix underflow

    }



    let lil_len = lil.len();

    for i in 0..=(big.len() - lil_len) {

        if big[i..i + lil_len] == lil[..] {

            return true;

        }

    }

    false

}
```

## [High] No Bounds Checking on Array Slicing in get_bps_init_data() - Panic-Based DoS

**Status:Fixed**

**Description** The `get_bps_init_data()` function performs multiple array slicing operations without verifying sufficient data exists. Lines 124-193 slice `bytes` array for: extr_root (32 bytes), parent_hash (32 bytes), state_root (32 bytes), pre_runtime (8 bytes), seal (64 bytes), and dynamically-sized extrinsics/events data. If `bytes` length is less than expected, Rust panics with "range end index out of bounds". Any user can create data PDAs via `init_data_pda()` which only checks if contract is paused, not authorization.

**Exploit Scenario**

1. Attacker creates malicious data PDA with only 50 bytes of data (needs ~160+ minimum)
2. Attacker calls `bps::init()` with this data PDA
3. Function reads first 8 bytes for block number (offset becomes 8)
4. Function attempts `&bytes[8..40]` for extr_root - succeeds (50 bytes available)
5. Function attempts `&bytes[40..72]` for parent_hash - panics (only 50 bytes total)
6. Program crashes with "range end index 72 out of range for slice of length 50"
7. All subsequent attempts to process this block fail permanently
8. If attacker targets legitimate block seed, real bridge operation is permanently blocked
9. Attacker can spam multiple malicious data PDAs, causing widespread bridge DoS
10. No recovery mechanism exists - that block seed is forever tainted

**Files**

- csov-bridge/src/bps.rs (lines 122-193)
- csov-bridge/src/data.rs (no validation on data content)

**Recommendation** Add bounds checking before all array slicing:

```
fn safe_slice<'a>(bytes: &'a [u8], offset: &mut usize, len: usize) -> Result<&'a [u8], ProgramError> {

    if *offset + len > bytes.len() {

        msg!("Insufficient data: need {} bytes at offset {}, but only {} total",

             len, *offset, bytes.len());

        return Err(ProgramError::InvalidInstructionData);

    }

    let slice = &bytes[*offset..*offset + len];

    *offset += len;

    Ok(slice)

}

// Then use throughout get_bps_init_data():

let extr_root = safe_slice(&bytes, &mut offset, 32)?;

let parent_hash = safe_slice(&bytes, &mut offset, 32)?;

let state_root = safe_slice(&bytes, &mut offset, 32)?;

let pre_runtime = safe_slice(&bytes, &mut offset, 8)?;

let seal = safe_slice(&bytes, &mut offset, 64)?;

// For dynamic slicing:

let extrinsic_len = usize_from_be_bytes(&bytes, &mut offset);

let extrinsic_bytes = safe_slice(&bytes, &mut offset, extrinsic_len)?;
```

## [High] No Bounds Checking in walk_merkle() - Panic-Based DoS

**Status:Fixed**

**Description** The `walk_merkle()` function at line 16 performs array slicing without validating that sufficient data exists: `&bytes[*offset..*offset + merkle_line_len]`. The `merkle_line_len` is read from untrusted data at line 15 with no validation that `*offset + merkle_line_len <= bytes.len()`. This causes Rust to panic with "range end index out of bounds" when processing malicious merkle proof data.

**Exploit Scenario**

1. Attacker creates malicious data PDA with merkle section
2. Sets `merkle_lines = 2` to trigger processing
3. Sets first `merkle_line_len = 50` (valid)
4. Sets second `merkle_line_len = 10000` (exceeds remaining bytes)
5. Calls `bps::init()` which eventually calls `walk_merkle()`
6. At line 16, program attempts `&bytes[offset..offset+10000]` when only 100 bytes remain
7. Program panics: "range end index out of bounds"
8. All subsequent attempts to process this block fail with panic
9. If attacker uses legitimate block seed, real bridge operation is permanently blocked
10. Attack costs attacker only ~0.002 SOL for data PDA rent

**Files**

- csov-bridge/src/merkle.rs (line 16)
- csov-bridge/src/bps.rs (line 204 - caller)

**Recommendation** Add bounds checking before slicing:

```
pub fn walk_merkle<'a>(

    bytes: &'a [u8],

    offset: &mut usize,

    state_root_arr: [u8; 32],

) -> (bool, &'a [u8]) {

    let merkle_lines = usize_from_be_bytes(bytes, offset);



    let mut prev_hash = [0u8; 32];

    let mut first_merkle_line: &[u8] = &[];

    for i in 0..merkle_lines {

        let merkle_line_len = usize_from_be_bytes(bytes, offset);



        // ADD: Bounds check

        if *offset + merkle_line_len > bytes.len() {

            msg!("Insufficient data for merkle line: need {} bytes", merkle_line_len);

            return (false, &[]);

        }



        let curr_merkle_line = &bytes[*offset..*offset + merkle_line_len];

        // ... rest of function

    }

}
```

## [Medium] No Validation on Merkle Line Count - Compute Exhaustion DoS

**Status:Fixed**

**Description** The `walk_merkle()` function reads `merkle_lines` count from untrusted data at line 10 without any maximum limit validation before looping. Attacker can provide huge counts (e.g., 100,000) causing the program to loop excessively, with each iteration reading data, slicing bytes, and computing blake2b hash (~2000 compute units per hash). This exhausts Solana's compute unit limit causing transaction failure and permanent DoS for that data PDA.

**Exploit Scenario**

1. Attacker creates data PDA with `merkle_lines = 100_000`
2. Attacker calls `bps::init()` which calls `walk_merkle()`
3. Line 14 starts loop: `for i in 0..100_000`
4. Each iteration: reads 8 bytes (line 15), slices array (line 16), computes blake2b_32 hash (line 25)
5. After ~700 iterations, transaction exceeds 1.4M compute unit limit
6. Transaction fails: "Program failed to complete: exceeded maximum number of instructions"
7. Block data cannot be processed - permanent DoS for this data PDA
8. If attacker targeted legitimate block seed, real bridge operation is blocked
9. Each attack costs attacker only ~0.002 SOL for data PDA rent

**Files**

- csov-bridge/src/merkle.rs (line 10, 14)
- csov-bridge/src/bps.rs (line 204)

**Recommendation** Add maximum limit based on realistic Polkadot merkle proof depth:

```
pub fn walk_merkle<'a>(

    bytes: &'a [u8],

    offset: &mut usize,

    state_root_arr: [u8; 32],

) -> (bool, &'a [u8]) {

    const MAX_MERKLE_LINES: usize = 100; // Reasonable for Polkadot merkle proofs



    let merkle_lines = usize_from_be_bytes(bytes, offset);

    if merkle_lines > MAX_MERKLE_LINES {

        msg!("Too many merkle lines: {}", merkle_lines);

        return (false, &[]);

    }



    // ... rest of function

}
```

## [Medium] No Validation on Merkle Line Length - Memory DoS

**Status:Fixed**

**Description** The `walk_merkle()` function reads `merkle_line_len` from untrusted data at line 15 without validating it's within reasonable bounds before attempting to slice that amount of data. Attacker can specify huge lengths (e.g., 10MB) which either causes panic when bounds are exceeded, or if within bounds, passes huge data to `blake2b_32()` causing excessive compute unit consumption.

**Exploit Scenario**

1. Attacker creates data PDA with merkle data where first `merkle_line_len = 1_000_000` (1MB)
2. Attacker calls `bps::init()` which calls `walk_merkle()`
3. Line 15 reads: `merkle_line_len = 1_000_000`
4. Line 16 attempts to slice 1MB of data (likely panics due to insufficient data)
5. OR if data exists, line 25 attempts to hash 1MB via `blake2b_32(curr_merkle_line)`
6. Blake2b hashing 1MB consumes excessive compute units
7. Transaction fails - DoS for this data PDA
8. Alternative: Set length just beyond available bytes to guarantee panic

**Files**

- csov-bridge/src/merkle.rs (lines 15-16)
- csov-bridge/src/bps.rs (line 204)

**Recommendation** Add maximum length validation:

```
for i in 0..merkle_lines {

    const MAX_MERKLE_LINE_LEN: usize = 10_000; // ~10KB per line



    let merkle_line_len = usize_from_be_bytes(bytes, offset);

    if merkle_line_len > MAX_MERKLE_LINE_LEN {

        msg!("Merkle line too large: {}", merkle_line_len);

        return (false, &[]);

    }



    if *offset + merkle_line_len > bytes.len() {

        return (false, &[]);

    }



    let curr_merkle_line = &bytes[*offset..*offset + merkle_line_len];

    // ... rest of loop

}
```

## [Medium] Duplicate Lock Processing via Malicious Event Data

**Status:Fixed**

**Description** The event processing loop in `get_bps_init_data()` at lines 207-224 does not prevent duplicate extrinsic IDs in success events. If malicious block data contains multiple success events claiming the same extrinsic ID, the same lock operation is added to `proven_locks` multiple times. This bypasses the intended one-lock-per-extrinsic guarantee and could cause multiple token mints for a single lock operation.

**Exploit Scenario**

1. Legitimate user locks 1000 CSOV on Polkadot (extrinsic ID = 5 in block)
2. Attacker creates malicious data PDA (no authorization check) with:
    - Legitimate extrinsics list including the lock at position 5
    - Success events array: [Success(extrinsic=5), Success(extrinsic=5), Success(extrinsic=5)]
    - Valid merkle proofs (attacker can construct these)
3. Attacker calls `bps::init()` which processes malicious data
4. Loop processes first success event for extrinsic 5 → adds lock to proven_locks (amount=1000 CSOV)
5. Loop processes second success event for extrinsic 5 → adds SAME lock again (amount=1000 CSOV)
6. Loop processes third success event for extrinsic 5 → adds SAME lock third time (amount=1000 CSOV)
7. `proven_locks` now contains same lock 3 times: [Lock(1000), Lock(1000), Lock(1000)]
8. BPS is created with 3 duplicate locks in `bps.locks` array
9. Attacker creates valid SRW for the Polkadot source address
10. During `finalize()`, all 3 locks are processed → vesting contract receives 3x the amount
11. User receives 3000 ORION tokens instead of 1000 ORION
12. Extra 2000 ORION tokens are minted from thin air (inflation attack)

**Files**

- csov-bridge/src/bps.rs (lines 207-224)
- csov-bridge/src/finalize.rs (processes proven_locks without duplicate checking)

**Recommendation** Track processed extrinsic IDs to prevent duplicates:

```
use std::collections::HashSet;

let mut proven_extrinsic_ids = HashSet::new();

for i in 0..successes_count {

    let success = successes[i];

    if !contains(first_merkle_line, success) {

        continue;

    }

    let proven_extrinsic_id = successes_to_extrinsic_id[i];



    // Skip if already processed this extrinsic ID

    if !proven_extrinsic_ids.insert(proven_extrinsic_id) {

        msg!("Warning: Duplicate success event for extrinsic ID {}", proven_extrinsic_id);

        continue;

    }



    for j in 0..lock_ops_idx.len() {

        if lock_ops_idx[j] == proven_extrinsic_id {

            proven_locks.push(ExtrinsicStatus {

                source: lock_ops[j].source,

                amount: lock_ops[j].amount,

                is_vested: lock_ops[j].is_vested,

            });

            break; // Only add once per extrinsic ID

        }

    }

}
```

## [Medium] No Validation on Count Values - Compute Unit Exhaustion DoS

**Status:Fixed**

**Description** The `get_bps_init_data()` function reads `extrinsics_count` and `successes_count` from untrusted data without upper bound validation before allocating vectors and looping. Attacker can provide huge counts (e.g., 1,000,000) causing the program to attempt massive allocations and exceed Solana's compute unit limit of ~1.4M units, causing transaction failure and permanent DoS for that data PDA.

**Exploit Scenario**

1. Attacker creates data PDA with `extrinsics_count` encoded as 10,000,000 (no authorization check)
2. Attacker calls `bps::init()` to process this data
3. Line 140 reads: `extrinsics_count = 10_000_000`
4. Line 145 starts loop: `for i in 0..10_000_000`
5. Each iteration allocates memory, performs slicing, calls `extrinsic_details()`, checks patterns

6. After ~10,000 iterations, transaction exceeds 1.4M compute unit limit
7. Transaction fails: "Program failed to complete: exceeded maximum number of instructions"
8. Block data cannot be processed - permanent DoS for this data PDA
9. If attacker used legitimate block seed, real bridge operation is blocked
10. Attacker can repeat for multiple blocks, causing widespread DoS
11. Each attack only costs attacker rent for data PDA (~0.002 SOL)

Alternative: `successes_count` can similarly cause DoS at line 187 with similar compute exhaustion.

**Files**

- csov-bridge/src/bps.rs (lines 140, 187)

**Recommendation** Add maximum limits based on realistic Polkadot block constraints:

```
// Add to constants.rs

pub const MAX_EXTRINSICS_PER_BLOCK: usize = 10_000;

pub const MAX_SUCCESS_EVENTS: usize = 10_000;

// In get_bps_init_data():

let extrinsics_count = usize_from_be_bytes(&bytes, &mut offset);

if extrinsics_count > MAX_EXTRINSICS_PER_BLOCK {

    msg!("Extrinsics count {} exceeds maximum {}", extrinsics_count, MAX_EXTRINSICS_PER_BLOCK);

    panic!("Too many extrinsics");

}

// Later:

let successes_count = usize_from_be_bytes(&bytes, &mut offset);

if successes_count > MAX_SUCCESS_EVENTS {

    msg!("Successes count {} exceeds maximum {}", successes_count, MAX_SUCCESS_EVENTS);

    panic!("Too many success events");

}
```

## [Medium] SRW PDA Can Be Overwritten Destroying Previous Bridge Requests

**Status:Fixed**

**Description** The `srw::init()` function allows overwriting existing SRW PDAs if status is Closed OR if status is Open but signature verification is incomplete (i < 1000). Line 110 unconditionally serializes new data into the PDA, destroying any previous bridge request state including signature verification progress. Only fully verified SRWs (status=Open AND i=1000) are protected from overwriting.

**Exploit Scenario** Assuming Alice is legitimately bridging 1000 CSOV tokens:

1. **Alice Initiates Bridge Request:**
    - Alice calls `srw::init()` with her DOT source address `DOT_ALICE`
    - SRW PDA created at address derived from `DOT_ALICE`
    - Alice pays 0.02 SOL bridge fee
    - SRW stores: signature for transferring to `SOL_ALICE`, status=Open, i=300
2. **Signature Verification In Progress:**
    - Backend calls `srw::check()` repeatedly, advancing i from 300 → 400 → 500...
    - After 10 transactions, i reaches 800 (getting close to completion)
    - Alice's signature is valid and will eventually reach i=1000
3. **Attacker Overwrites SRW:**
    - Attacker (could be same user or different user) calls `srw::init()` with SAME DOT source `DOT_ALICE`
    - Attacker provides different Solana target `SOL_ATTACKER` and different signature
    - System checks: SRW exists and status=Open, but i=800 (not 1000 yet)
    - Attacker pays 0.02 SOL fee
    - SRW data is OVERWRITTEN: signature changes to attacker's signature, i resets to 300, message changes to `SOL_ATTACKER`
4. **Alice's Request Is Lost:**

1. **Alice's Request is Lost:**
    - Alice's original signature and verification progress (i=800) is completely erased
    - Alice's 0.02 SOL fee was already paid and is not refundable
    - Alice cannot recover her bridge request
    - If backend continues verification, it's now verifying ATTACKER's signature, not Alice's
    - When finalization happens (if ever), tokens go to `SOL_ATTACKER`, not `SOL_ALICE`
5. **Griefing Attack Variant:**
    - Attacker monitors for SRW init transactions
    - Immediately calls `srw::init()` with same DOT address but invalid signature
    - Resets verification progress and wastes backend's gas on invalid signature
    - Can repeat this indefinitely, preventing any bridge completion for that DOT address
    - Each grief costs attacker 0.02 SOL but blocks the DOT address from bridging
6. **Front-Running Attack Variant:**
    - User Alice submits legitimate bridge request for 1000 CSOV
    - Attacker sees transaction in mempool
    - Attacker front-runs with `srw::init()` for same DOT address but attacker's Solana address
    - Both transactions succeed (Alice's initializes, attacker's overwrites)
    - Alice paid 0.02 SOL but her bridge goes to attacker's address

**Files**

- csov-bridge/src/srw.rs
- csov-bridge/src/enums/srw_status_enum.rs

**Recommendation** Prevent overwriting of any existing SRW PDAs. Modify the init() function to reject requests if an SRW already exists for that DOT source address, regardless of status:

```
// In srw::init() - Add check after line 42

if !srw_pda.data_is_empty() || srw_pda.lamports() > 0 {

    return Err(ERR_SRW_ALREADY_EXISTS);

}

Optionally, add a cleanup instruction to close SRWs that have been finalized or failed verification, allowing address reuse:

pub fn close_srw(program_id: &Pubkey, accounts: &[AccountInfo]) -> ProgramResult {

    let srw_struct = utils::client_srw_struct(srw_pda);



    if srw_struct.status != SrwStatus::Closed &&

       srw_struct.dot_sig_check_struct.i != DOT_SIG_CHECK_I_KO {

        return Err(ERR_CANNOT_CLOSE_ACTIVE_SRW);

    }



    // Transfer rent back and close account

    **beneficiary.lamports.borrow_mut() += srw_pda.lamports();

    **srw_pda.lamports.borrow_mut() = 0;

    Ok(())

}
```

## [High] Unbounded Loop in release() Causes Compute Limit DoS

**Status:Fixed**

**Description** The `release()` function at line 98 loops through ALL vesting entries in the vesting list without any limit: `for i in 0..len`. Each iteration reads data, performs timestamp calculations, and potentially calls `mint()` which performs CPI to SPL token program (lines 188-229). If the vesting list contains hundreds or

thousands of entries, the transaction will exceed Solana's ~1.4M compute unit limit and fail. As more vesting entries are added over time, `release()` becomes permanently unusable.

**Exploit Scenario**

1. Bridge operates normally for 6 months, adding 1 vesting entry per bridge request
2. 1000 users bridge tokens, creating 1000 vesting entries in the list
3. User calls `release()` to claim vested tokens after cliff period
4. Line 98 starts loop: `for i in 0..1000`
5. Each iteration: timestamp calculation (~~100 CU), borsh serialize update (~~200 CU), mint() CPI (~5000 CU per entry that needs minting)
6. After ~200 iterations, transaction exceeds 1.4M compute unit limit
7. Transaction fails: "Program failed to complete: exceeded maximum number of instructions"
8. NO user can release ANY vested tokens - entire vesting contract becomes unusable
9. Funds are permanently locked (contract still owns mint authority)
10. No recovery mechanism exists - would require contract upgrade

**Files**

- orion-vesting/src/release.rs (lines 97-143)

**Recommendation** Redesign to allow partial releases or per-user vesting tracking:

**Option 1: Per-user vesting PDA**

```
// Instead of single vesting list PDA, use per-user PDAs
// Seed: [b"vesting", source_address]
// This way each user's release only loops through THEIR entries
pub struct UserVestingList {
    pub entries: Vec<VestingEntryStruct>,
}
// In release():
pub fn release(program_id: &Pubkey, accounts: &[AccountInfo]) -> ProgramResult {
    // ... validation ...
    let user_wallet = next_account_info(accounts_iter)?;
    let user_vesting_pda = next_account_info(accounts_iter)?;

    // Verify PDA matches user
    let (computed, _) = Pubkey::find_program_address(
        &[b"vesting", user_wallet.key.as_ref()],
        program_id
    );
    if computed != *user_vesting_pda.key {
        return Err(ERR_INVALID_USER_VESTING_PDA);
    }

    // Now only loop through this user's entries (typically 1-10 entries)
    let mut user_vesting: UserVestingList =
        UserVestingList::try_from_slice(&user_vesting_pda.data.borrow())?;

    for entry in user_vesting.entries.iter_mut() {
        // ... calculate and mint ...
    }
}
```

**Option 2: Pagination**

```
pub fn release(

    program_id: &Pubkey,

    accounts: &[AccountInfo],

    instruction_data: &[u8],

) -> ProgramResult {

    // instruction_data contains: [start_index (4 bytes), count (4 bytes)]

    let start_idx = u32::from_be_bytes(instruction_data[1..5].try_into().unwrap()) as usize;

    let count = u32::from_be_bytes(instruction_data[5..9].try_into().unwrap()) as usize;



    const MAX_ENTRIES_PER_TX: usize = 10;

    if count > MAX_ENTRIES_PER_TX {

        return Err(ProgramError::InvalidInstructionData);

    }



    let end_idx = std::cmp::min(start_idx + count, vesting_list.list.len());



    for i in start_idx..end_idx {

        // ... process entries ...

    }

}
```

## [High] release() Requires All Unvested Users' Wallets - Multi-User Coordination Impossible

**Status:Fixed**

**Description** The `release()` function loops through ALL vesting entries (line 98) and attempts to mint for each unvested entry. The `mint()` helper function (lines 162-171) searches for each entry's target wallet in the provided accounts. If ANY unvested entry's target wallet is missing from the accounts, the transaction fails with `ERR_NO_WALLET_AND_ATA_FOR_TARGET`. This makes it impossible for individual users to release their own tokens when other users also have unvested tokens, as each user would need to coordinate and provide wallet accounts for ALL other unvested users.

**Exploit Scenario**

1. Bridge operates for 3 months, 10 users bridge tokens creating 10 vesting entries
2. After 13 months (past 12-month cliff), all 10 users have some tokens vested
3. Alice wants to release her 100 vested tokens
4. Alice calls `release([Alice_wallet, Alice_ATA])` providing only her own accounts
5. Loop at line 98 starts processing all 10 entries:
    - Entry 0 (Alice): Line 130 calls mint() → finds Alice's wallet → mints 100 tokens → SUCCESS
    - Entry 1 (Bob): Line 104 calculates Bob has vested tokens (not skipped) → Line 130 calls mint()
    - mint() line 164-167: Searches for Bob's wallet in provided accounts → NOT FOUND
    - Line 170: Returns `ERR_NO_WALLET_AND_ATA_FOR_TARGET`
    - Line 142: `?` propagates error, transaction FAILS
6. Transaction reverts completely (Solana atomicity) - Alice's tokens NOT released
7. Alice cannot release her own tokens unless she:
    - Provides wallet/ATA accounts for ALL 9 other unvested users (impossible - she doesn't control their wallets), OR
    - Waits 48 months until all users are fully vested (entries skipped at line 99-102), OR
    - Never releases (funds locked)
8. Same problem affects ALL users - nobody can release independently
9. Contract becomes completely unusable for multi-user scenarios

**Files**

- orion-vesting/src/release.rs (lines 98-143 main loop, lines 162-171 wallet search in mint())

**Recommendation** This requires fundamental redesign. The release() function MUST allow per-user releases:

**Option 1: Per-user vesting PDAs (RECOMMENDED)**

```
 // Change from single global list to per-user PDAs

// Seed: [b"vesting", target_address]

pub fn release(program_id: &Pubkey, accounts: &[AccountInfo]) -> ProgramResult {

    let accounts_iter = &mut accounts.iter();

    let payer = next_account_info(accounts_iter)?;

    // ... system accounts ...

    let user_wallet = next_account_info(accounts_iter)?;  // User's wallet

    let user_vesting_pda = next_account_info(accounts_iter)?;



    // Verify PDA is for this user

    let (computed, _) = Pubkey::find_program_address(

        &[b"vesting", user_wallet.key.as_ref()],

        program_id

    );

    if computed != *user_vesting_pda.key {

        return Err(ERR_INVALID_USER_VESTING_PDA);

    }



    // Now only process THIS user's entries (typically 1-5 entries)

    let mut user_vesting: UserVestingList =

        UserVestingList::try_from_slice(&user_vesting_pda.data.borrow())?;



    for entry in user_vesting.entries.iter_mut() {

        // Calculate and mint only for THIS user

        // Only need THIS user's wallet/ATA in accounts

    }

}
```

**Option 2: Add target filter parameter**

```
pub fn release(

    program_id: &Pubkey,

    accounts: &[AccountInfo],

    instruction_data: &[u8],  // Contains target address to release for

) -> ProgramResult {

    // Parse target from instruction_data

    let target_to_release = Pubkey::new_from_array(

        instruction_data[1..33].try_into().unwrap()

    );



    // Only process entries matching this target

    for i in 0..len {

        let entry_target = Pubkey::new_from_array(vesting_list.list[i].target);

        if entry_target != target_to_release {

            continue;  // Skip entries for other users

        }

        // Process only this user's entries

    }

}
```