Security Research Labs

Serai Baseline Security Assurance

Threat model and hacking assessment report

v1.0, 01 April 2026

Prepared for:
Serai DEX

- Public - 60 pages -

Content

Version	v1.0
Client	Serai
Date	01 April 2026
Assessment Team	Jonas Panizza Nils Ollrogge Gabriel Arnautu Marc Heuse

Version History

Date	Version
March 31, 2026	v0.1 – draft release
April 01, 2026	v1.0 – final report

Disclaimer

This report describes the findings and core conclusions derived from the audit carried out by Security Research Labs within the timeframe and scope detailed in Chapter 3.

Please note that this report does not guarantee that all existing security vulnerabilities have been exhaustively identified. Implementing the recommendations provided herein does not ensure that future code will remain free of bugs.

Integrity Notice

Assessment Timeline

Security Research Labs performed the Serai source code security assessment over the course of 11 weeks, starting from 12th of January 2026.

Date	Event
January 12, 2026	Assessment kick-off
March 31, 2026	Preliminary findings report
April 01, 2026	Final report

Table 1: Audit timeline

1 Executive summary

1.1 Engagement overview

Security Research Labs is an established provider of specialized security audit services, founded in 2010, providing expert blockchain audit services since 2019.

This report documents the results of a security assurance audit of the Polkadot SDK based components of Serai that Security Research Labs performed between January to March 2026. During this study, the developers of Serai provided access to relevant documentation and effectively supported the research team. We verified the protocol design, concept documentation, and relevant available source code of Serai.

This audit focused on assessing Serai's codebase for resilience against hacking and abuse scenarios. Key areas of scrutiny included all substrate pallets, the serai node and runtime, the build pipeline and the custom patches to Polkadot SDK that the Serai team created. The testing approach combined static and dynamic analysis techniques, leveraging both automated tools and manual inspection. We prioritized reviewing critical functionalities and conducting thorough security tests to ensure the robustness of Serai's codebase and the theory behind it. We collaborated closely with the developers, utilizing full access to source code and documentation to perform a rigorous assessment. The communication was excellent and solution-oriented, the team was highly responsive to even the most minor reported issues.

This independent security audit was funded by the MAGIC Grants charity.

1.2 Observations and risk

The research team identified several issues ranging from high to informational level severity. Most of the identified issues were DoS vectors enabling attackers to trigger panics in the blockchain runtime.

The development team of Serai acknowledged these issues and is actively working on remediation in cooperation with us. By now, most of the identified issues have been successfully remediated.

1.3 Recommendations

To further improve the security of the Serai DEX, we recommend introducing a constrained on-chain governance and upgrade mechanism to enable rapid incident response, minimizing divergence from the upstream Polkadot SDK to reduce long-term security and maintenance risks, and introducing dynamic analysis practices (e.g., fuzzing of critical components) to improve the detection of subtle logic bugs and runtime panics.

2 Evolution suggestions

To ensure that Serai is secure against further unknown or yet undiscovered threats, we recommend considering the following evolution suggestions and best practices described in this section.

2.1 Business logic improvement suggestions

Add governance support and programmatic runtime upgrades. While avoiding sudo removes a privileged control path, the lack of an on-chain upgrade mechanism limits the chain's ability to respond quickly to malicious activity or urgent situations. As a result, upgrades must be coordinated off-chain, increasing operational complexity and slowing incident response. Introducing a constrained governance and upgrade mechanism would improve maintainability and recovery without requiring unrestricted administrative control.

2.2 Secure development improvement suggestions

We recommend further strengthening the security of the blockchain by implementing the following best practices:

Adopt pair programming. Pair programming practices help identify security vulnerabilities early in the development cycle. This is beneficial for better code security because it involves immediate code review, allowing vulnerabilities to be caught in real time. Pair programming fosters continuous knowledge transfer reduces the risk of oversights and encourages adherence to coding standards. This collaborative approach also boosts problem solving, morale, and accountability, ensuring early detection of security issues and fostering continuous learning and improvement.

Implement security checks close to business logic. Security checks and access controls should be implemented near the relevant logic, rather than in distant or unrelated parts of the codebase. This makes security controls easier to validate and reduces the risk that changes in one part of the system will have unintended security effects on another. One example is the authorization for the slash_serai_validator extrinsic, which is enforced in a completely different location within wasm/mod.rs.

Perform dynamic analysis. Developing fuzzing harnesses for consensus mechanisms, domain-specific areas, and other critical components is essential for identifying security vulnerabilities and business logic issues. By employing invariants, these fuzzing tests can effectively uncover subtle flaws that might otherwise go unnoticed. The Polkadot codebase exemplifies this approach by utilizing multiple fuzzing harnesses based on the substrate-runtime-fuzzer, our in-house developed template for the Polkadot SDK ecosystem. This demonstrates how comprehensive and targeted fuzz testing can significantly enhance the security and reliability of complex systems.

Create an incident response plan. Developing a comprehensive incident response plan to address potential security breaches is vital for maintaining code security and organizational resilience. This plan should include detailed procedures for responding to various scenarios, such as compromised developers or exploited blockchain vulnerabilities, to ensure quick and effective mitigation of threats. By having a well-defined response strategy, organizations can minimize the impact of security incidents, protect sensitive data, and maintain trust with users and stakeholders.

2.3 Further recommended best practices

Extensive test implementation. Serai has pallet-level unit tests across much of its Substrate code, but coverage is not exhaustive. There are multiple places in the code, for example in the validator-sets tests and genesis-liquidity tests that contain TODOs indicating gaps in test coverage. We recommend extending unit tests to provide a stronger focus on security. This includes testing for edge cases, potential vulnerabilities, and common attack vectors.

Documentation. The documentation was occasionally incomplete or slightly outdated, leading to inconsistencies with the code. This made it challenging to understand the code without consulting the development team. Accurate documentation ensures that developers, auditors, and other

stakeholders can comprehend the codebase without frequently consulting the development team. To achieve this, establish a practice of updating the documentation concurrently with any code changes. Incorporating documentation verification into the code review process can help detect discrepancies early.

3 Motivation and scope

Serai is a decentralized cross chain exchange (DEX) built on the Polkadot SDK that supports swaps to and from Bitcoin, Monero, and Ethereum. At a high level, it combines a custom blockchain runtime with substantial off-chain infrastructure to offer automated market maker trading through liquidity pools, where prices are determined algorithmically rather than by an order book. Users can initiate swaps by depositing supported assets from external chains into Serai-controlled addresses, after which the activity is recorded on Serai, executed against the relevant liquidity pools, and settled in the requested asset.

While Serai uses the Polkadot SDK as its foundation, it aims to remain as independent as possible from the underlying framework. To support this goal, the codebase introduces a broad abstraction layer on top of Substrate, applies targeted custom patches to the Polkadot SDK, and uses Borsh serialization instead of SCALE.

The core business logic of Serai centers on validator sets, liquidity pools, and cross chain asset handling. Validator sets allocate stake to secure the assets represented inside Serai's pools, forming the basis of Serai's economic security model, and use threshold signatures to control wallets on external chains. This is complemented by off-chain services that monitor Bitcoin, Monero, and Ethereum for relevant transactions, interpret deposit instructions, and coordinate the signing and submission of transactions to Serai that include data describing external chain activity.

On the technical level, the Serai blockchain itself is implemented in Rust and follows the standard architecture of Polkadot SDK based networks, with a WebAssembly runtime, peer to peer communication via libp2p, and a block production engine. Serai's runtime consists of multiple modules compiled into a WASM blob stored on chain and executed by nodes as the canonical definition of the protocol's rules.

Security Research Labs collaborated with the Serai team to create an overview containing the components in scope and their audit priority. **For this audit, only the Polkadot SDK based components of the Serai DEX were in-scope.** The components and their assigned priorities are reflected in Table 2.

Repository	Priority	Component(s)	Reference
https://github.com/serai-dex/serai	High	abi, primitives, core, median	786ba87
	High	coins, signals pallet	24eefa8
	High	dex pallet	30e61ad
	High	genesis-liquidity, economic-security pallet	391f1f0
	High	validator-sets pallet	c079232
	High	emissions pallet	85b9d79
	High	in-Instructions pallet	e962cf4
	Medium	runtime	e962cf4
	Medium	build pipeline	3f5d734

	Medium	node	cf4276f
https://github.com/serai-dex/patch-polkadot-sdk	Low	patch-polkadot-sdk	202d171

Table 2: In-scope Serai's components with audit priority

4 Methodology

The methodology used in this assessment follows a structured, attacker-centric approach aimed at identifying flaws in the Serai project. The methodology consists of four steps: (1) threat modelling, (2) security design coverage checks, (3) implementation baseline check, and finally (4) remediation support. The threat modelling was conducted in alignment with the STRIDE threat modelling framework.

4.1 STRIDE threat modelling framework

The STRIDE framework is a widely used security model that helps identify potential threats in software systems. It stands for Spoofing, Tampering, Repudiation, Information disclosure, Denial of Service, and Elevation of privilege, each representing a different category of security risks.

Spoofing

Spoofing refers to bugs or logical flaws that cause the application to misattribute a transaction to a different user. This could be caused for example due to signature verification issues or missing input validation. As a worst-case scenario, this would allow a user to have access to tokens that they do not own.

Tampering

Tampering in Serai involves the manipulation of data without proper authorization or security controls. This could lead to a user influencing the validator set or creating malicious blocks that would be accepted.

Repudiation

Repudiation threats in the context of Serai relate to the ability of an attacker to deny having performed certain actions or not being punished by the system when misbehaving.

Information disclosure

Information disclosure occurs when sensitive data can be used to gain financial advantage, as in executing a sandwich attack on the liquidity pools.

Denial of service (DoS)

Denial-of-service attacks in the context of Serai aim to restrict application access for legitimate users. To achieve this, attackers might use application behavior that triggers high resource consumption to exhaust computational resources and impact application availability for all users.

Elevation of privilege

Privilege escalation attacks occur when an adversary can gain persistent access to roles that are more capable than their original role. This could be achieved by compromising the distribution channel of the WASM runtime.

4.2 Threat modeling and attacks

The goal of the threat model framework is to determine specific areas of risk for Serai, including both potential threats and concrete attack scenarios. Familiarity with these risk areas can provide guidance for the design of the implementation stack, the actual implementation of the stack, as well as security testing.

Our threat modelling consists of 3 stages, each building upon the previous one, resulting in a list of attacks and their severities, graded from low to critical.

In the first stage, each STRIDE category is assigned a *STRIDE Hacking Value* which measures how valuable a threat in each category is to an attacker. The *STRIDE Hacking Value* is categorized into low, medium or high, with the following definitions:

- **Low:** Threats in this STRIDE category offer the hacker little to no gain.

- **Medium:** Threats in this STRIDE category offer the hacker considerable gains.
- **High:** Threats in this STRIDE category offer the hacker high gains.

In the second stage, concrete threat scenarios are identified and mapped to their corresponding STRIDE categories. Each threat is assigned a *Threat Impact* value, which represents the potential damage the threat could cause to the project. *Threat Impact* is categorized into low, medium or high, with the following definitions:

- **Low:** Threat scenario would cause negligible damage to the project.
- **Medium:** Threat scenario poses a considerable threat to the project.
- **High:** Threat scenario poses an existential threat to the project.

Based on the *Threat Impact* and the *STRIDE Hacking Value* of the associated category, a *Threat Risk* is derived according to Table 3. The *Threat Risk* represents the overall risk posed by each identified threat.

STRIDE Hacking Value/Threat impact	Low	Medium	High
Low	Low	Medium	Medium
Medium	Medium	Medium	High
High	Medium	High	High

Table 3: Threat Risk calculation matrix

In the third and final stage, concrete attack scenarios are identified and mapped to the previously defined threats. Each attack is assigned an *Attack Feasibility*, which measures how likely an attack is to succeed, considering complexity, required skill, and resources. *Attack Feasibility* is categorized into low, medium or high, with the following definitions:

- **Low:** Attack requires significant expertise and uncommon conditions.
- **Medium:** Attack requires reasonable effort and standard attacker capabilities.
- **High:** Attack requires minimal effort or widely available tools.

Based on the *Attack Feasibility* and the *Threat Risk* of the associated threat, an *Attack Severity* is derived according to Table 4.

Threat Risk / Attack Feasibility	Low	Medium	High
Low	Low	Medium	Medium
Medium	Medium	Medium	High
High	Medium	High	Critical

Table 4: Attack Severity calculation matrix

Together, these stages define a structured approach that progresses from abstract STRIDE threat categories and their value to an attacker, through concrete threat scenarios and their associated risk, and finally to specific attack scenarios with assigned severities.

Applying the framework to Serai, different areas of risk were identified. Table 5 provides a high-level overview of the identified hacking risks relevant to the application, including each STRIDE category with its associated hacking value, example threat scenarios, their derived threat risk, and corresponding example attacks.

The complete set of identified threat scenarios, together with the attacks that enable them, is documented in the threat model deliverable. This list can serve as a starting point for developers when assessing security considerations for future feature development. By thinking in terms of threat scenarios and attacks during code review or feature ideation, many issues can be caught or even avoided altogether.

Scope Category	STRIDE Hacking Value	Example threat scenarios	Threat Risk	Example Attack Ideas
Spoofing	High	Abuse ABI serialization problems to trigger unintended actions.	Medium	After a runtime upgrade changes the ABI call encoding, a client still on the old ABI submits bytes that decode on-chain as a different call or fail to decode, causing unintended execution or rejection/DoS.
Tampering	High	Influence a validator set.	High	An attacker controls a blocking minority of the Serai validator-set (33% = 43 shares) and denies participating.
Repudiation	High	Execute malicious transactions without being punished.	High	Abuse a bug in the custom patches applied by Serai to polkadot-sdk.
Information disclosure	Medium	Manipulate economic parameters and pricing.	Low	Leverage public information about transfers to liquidity pools to gain some kind of financial advantage (JIT liquidity, sandwich attacks).
Denial of service	Medium	Perform DoS attacks to halt or slow down block production.	Medium	An attacker is able to arbitrarily panic the runtime by calling an extrinsic.
Privilege escalation	High	Perform a supply chain attack to compromise the Serai network.	High	An attacker compromises the distribution channel of the WASM runtime and serves a malicious runtime during a runtime upgrade, gaining arbitrary code execution on nodes and enabling theft of all processed data.

Table 5: Risk overview

4.3 Security design coverage check.

Next, we reviewed the Serai design for coverage against relevant hacking scenarios. For each scenario, we have investigated the following two aspects:

a. **Coverage**. Is each potential security vulnerability sufficiently covered by our audit?
b. **Underlying assumptions**. Which assumptions must hold true for the design to effectively reach the desired security goal?

4.4 Implementation check

As a third step, we tested the current Serai implementation for openings whereby any of the defined hacking scenarios could be executed.

To effectively review the Serai codebase, we derived our code review strategy based on the threat model that we established as the first step. For each identified threat, hypothetical attacks were developed and mapped to their corresponding threat category, as outlined in Chapter 4.2.

Prioritizing potential risk for the network, the code was assessed for present protections against the respective threats and attacks as well as the vulnerabilities that make these attacks possible. For each threat, we:

1. Identified the relevant parts of the codebase, for example, the relevant pallets and the runtime configuration
2. Identified viable strategies for the code review. We performed manual code audits, fuzz testing, and manual tests where appropriate
3. Ensured the code did not contain any vulnerabilities that could be used to execute the respective attacks. Otherwise, we ensured that sufficient protection measures against specific attacks were present
4. Immediately reported any vulnerability that was discovered to the development team along with suggestions around mitigations

We carried out a hybrid strategy utilizing a combination of code review, static tests, and dynamic tests (e.g., fuzz testing) to assess the security of the Serai codebase.

While static and dynamic testing establishes a baseline assurance, the focus of this audit was on manual code review of the Serai codebase to identify logic bugs, design flaws, and best practice deviations. We reviewed the different components of Serai at multiple commits, as the codebase was still in development when the audit began, checking them whenever the developers indicated they were ready. We aimed to trace the intended functionality of the runtime modules in scope and to assess whether an attacker can bypass/misuse/abuse these components or trigger unexpected behavior on the blockchain due to logic bugs or missing checks. Since the Serai codebase is entirely open source, it is realistic that an adversary could analyze the source code while preparing an attack.

Fuzz testing is a technique to identify issues in code that handles untrusted input. In Serai‘s case these are extrinsics in the runtime.

Fuzz testing works by taking some valid input for a method under test, applying a semi-random mutation to it, and then invoking the method under test again with this semi-valid input. Through repeating this process, fuzz testing can unearth inputs that would cause a crash or other undefined behavior (e.g., integer overflows) in the method under test. The fuzz testing methods written for this assessment utilized the test runtime Genesis configuration as well as mocked externalities to execute the fuzz test effectively against the extrinsics in scope.

4.5 Remediation support

The final step is supporting Serai with the remediation process of the identified issues. Each finding was documented and published with mitigation recommendations. Once the mitigation solution is implemented, the fix is verified by us to ensure that it mitigates the issue and does not introduce other bugs.

During the audit, findings were shared via Serai's GitHub repository. We also used a private Signal channel for asynchronous communication and status updates. Additionally, we held jour fixe meetings to plan work and address any open questions regarding the codebase or the underlying theory behind Serai.

5 Static analysis assessment

We utilized Semgrep, an open-source static analysis framework that enables lightweight, pattern-based matching on source code syntax, to detect common classes of vulnerabilities in Substrate-based codebases. Specifically, at the kick-off of our manual review phase we applied a set of custom-developed Semgrep rules designed to identify misconfigurations in runtimes, unsafe logic constructs, and anti-patterns frequently encountered in Substrate development.

Upon completion of manual review phases for each feature, we further subjected the codebase to analysis using a curated set of reasoning-capable large language models (LLMs). These models were selected and benchmarked for their performance in identifying nuanced security risks and latent attack surfaces, thereby serving as a safeguard against human oversight or tunnel vision.

This methodology aims to complement and bolster the manual audit process by encapsulating it between two automated analysis stages: an initial rule-based scan to rapidly surface straightforward vulnerabilities, and a final LLM-driven assessment to highlight areas warranting deeper manual scrutiny.

6 Dynamic analysis assessment

During the audit, we utilized fuzz testing to identify bugs in the runtime of Serai. By applying fuzz testing, we aim to uncover issues that may go undetected through manual code review. We created a single harness with different configurations to fuzz the three stages that the Serai network can be in: genesis, post-genesis without economic security, and post-genesis with economic security. We tested all three configurations simultaneously by running three separate fuzzing campaigns. The fuzzing campaigns were continuously updated as new features were developed, ensuring that each new functionality was thoroughly tested for potential bugs and vulnerabilities. The campaign revealed three issues in Serai's codebase that could trigger a runtime panic.

Fuzz harness creation: We chose Ziggy, an open-source tool developed in-house, as our fuzzing orchestration tool. As for the harness, we used a customized harness for the runtime based on the substrate-runtime-fuzzer.

Coverage analysis and optimization: Although modern fuzzers can achieve good coverage by utilizing various techniques, we manually generated some seeds to target specific functionalities that were not covered by the fuzzer after a certain period. This approach assisted the fuzzer and optimized the overall coverage, ensuring more comprehensive testing. In our coverage analysis below, we only measure the Serai codebase coverage that the harness is targeting.

As shown in Table 6, the fuzzer achieved good coverage across the key components in scope for this audit. Lower coverage in certain parts of the codebase, such as the genesis liquidity, emissions, and in-instructions pallets, can be attributed to their reliance on complex state, lifecycle transitions, and externally derived inputs (e.g., signatures or time-based conditions), which are not fully modeled by the current harness.

Improving coverage in these areas would involve extending the harness to better simulate evolving state, realistic initialization, and structured inputs. This would enable deeper exploration of logic that is currently only partially exercised.

6.1 Runtime fuzzer

Component	Code Path	Coverage Achieved
Core crate	/substrate/core	70.87%
Primitives crate	/substrate/primitives	54.35%
ABI crate	/substrate/abi	40.00%
Median crate	/substrate/median	65.37%
Coins pallet	/substrate/coins	96.36%
Signals pallet	/substrate/signals	82.59%
Dex pallet	/substrate/dex	97.04%
Genesis liquidity pallet	/substrate/genesis-liquidity	32.24%
Economic security pallet	/substrate/economic-security	74.05%
Validator sets pallet	/substrate/validator-sets	62.29%
Emissions pallet	/substrate/emissions	39.43%

In-instructions pallet	/substrate/in-instructions	30.69%

Table 6: Code coverage achieved by runtime fuzzer

7 Findings summary

We identified 21 issues during our analysis of the runtime modules in scope in the Serai codebase that enabled some of the attacks outlined above. In summary, we found 7 high-severity, 6 medium-severity, 4 low-severity, and 4 information-level issues. An overview of all findings can be found in Table 7.

Critical	0	
High	7	
Medium	6	
Low	4	
Informational	4	
Total Issues	21	

Note: In our methodology, "critical-severity issues" are high-severity issues that could be exploited immediately by an attacker on already deployed infrastructure, including a parachain or an incentivized chain.

7.1 Issue summary

ID	Issue	Severity	Status
S3-727	serai-primitives: Panic during SCALE decoding when Borsh deserialization fails after successful reads	High	Mitigated
S3-729	serai-primitives: Panic on invalid public key during muSig key aggregation	High	Mitigated
S3-741	Pre-economic-security liquidity addition enables protocol-owned liquidity extraction	High	Mitigated
S3-753	`set_keys` validation vs dispatch mismatch	High	Mitigated
S3-758	Inverted GRANDPA equivocation proof validation allows bypass and false slashing	High	Mitigated
S3-759	Extra `floor(chunk / N)` transfer on resolve block can overdraw pending liquidity and trigger panic	High	Mitigated
S3-760	Inconsistent rounding between queue and resolve can cause panic	High	Mitigated
S2-733	serai-abi: Inverted nonce logic in transaction validation	Medium	Mitigated

S2-744	Unbounded State Growth via Zero Fee Dust Accounts	Medium	Acknowledged
S2-746	`quote_for_out` function in dex module may panic	Medium	Mitigated
S2-761	Zero drip rate causes pending liquidity chunks to remain unresolved	Medium	Mitigated
S2-763	SwapOut Allows Outbound Actions On Halted Destination Networks	Medium	Mitigated
S2-771	Expired external-network support can still be counted toward signal thresholds	Medium	Mitigated
S1-742	build.rs artifact selection is non-deterministic	Low	Mitigated
S1-745	`decrease_allocation` function in validator-sets sessions module may panic	Low	Mitigated
S1-747	Runtime panic due to multiplication overflow inside `sri_value` function	Low	Mitigated
S1-772	Retirement-signal favors become unremovable after signal expiry or revocation	Low	Acknowledged
S0-731	serai-primitives - Missing validation of External Coin and Network association	Info	Mitigated
S0-757	Genesis liquidity signature verification uses latest auxiliary keys instead of set-bound keys	Info	Mitigated
S0-774	Permissionless sweep of residual balances via zero-amount removal	Info	Risk accepted
S0-776	Unreviewed Security-Relevant TODOs	Info	Open

Table 7: Findings overview

8 Detailed findings

8.1 S3-727: serai-primitives: Panic during SCALE decoding when Borsh deserialization fails after successful reads

Attack scenario	**An attacker wants to perform DoS attacks to halt or slow down block production**
Classification	VLN-7: Reachable Panic
Component	primitives crate
Tracking	https://github.com/serai-dex/serai/issues/727
Attack impact	T-7: An attacker is able to arbitrarily panic the runtime by calling an extrinsic
Severity	High
Status	Mitigated

Background

The Serai primitives define base level functionality for the DEX.

Issue description

In serai-primitives in `src/sp_borsh/mod.rs:decode()`, the `borsh_as_scale!` adapter assumes that any `BorshDeserialize` error must originate from a failure in the underlying SCALE input. It stores the last SCALE read error in an internal field (`reader.1`) and, on Borsh failure, unconditionally unwraps that stored error. This assumption is however incorrect.

Borsh deserialization can fail after all bytes have been read successfully, for example when performing semantic or canonicality checks. From `src/sp_borsh/bitvec.rs:BitVec::deserialize_reader()`:

```
    let bits = usize::try_from(bits).map_err(|_| Error::other(LENGTH_ERROR))?;
    ...
    let mut result = BitVecInner::try_from_vec(result).map_err(|_|
Error::other(LENGTH_ERROR))?;
    ...
    if result.as_bitslice().iter().skip(bits).any(|bit| *bit) {
      Err(Error::other("non-canonical bit vector due to unused bits being
set"))?;
    }
```

In `src/sp_borsh/mod.rs:decode()` in such cases no SCALE read error exists, the stored error in `reader.1` remains `None`, and the `expect()` triggers a panic:

```
        <Self as borsh::BorshDeserialize>::deserialize_reader(&mut reader)
          .map_err(|_| reader.1.expect("Reader::read errored but didn't set the
error"))
```

This can be reproduced with valid, fully readable SCALE input that violates Borsh-level invariants, such as a non-canonical bit vector with unused bits set.

The following test added to `src/sp_borsh/mod.rs` will trigger the panic:

```
#[cfg(all(test, feature = "scale"))]
mod panic_tests {
  use super::BitVec;
  use scale::{Decode, Encode};

  use ::bitvec::{bitvec, order::Lsb0};

  #[test]
```

```
  fn non_canonical_bitvec_scale_decode_panics_until_fixed() {
    // 9 bits => 2 bytes; last byte has 7 unused bits (in Lsb0).
    let inner: ::bitvec::vec::BitVec<u8, Lsb0> = bitvec!(u8, Lsb0; 0; 9);
    let v = BitVec::<64>::try_from(inner).unwrap();

    // `borsh_as_scale!` encodes as Borsh bytes; mutate to make it non-
canonical:
    // set an unused bit in the last byte.
    let mut bytes = v.encode();
    *bytes.last_mut().unwrap() |= 0x80;

    // BUGGY: panics inside `borsh_as_scale!` Decode error-mapping (`reader.1`
is None).
    // FIXED: returns Err, and the assertion passes.
    let res = BitVec::<64>::decode(&mut &bytes[..]);
    assert!(res.is_err());
  }
}
$ cargo test --features scale
...
test sp_borsh::srlabs::trigger_decode_panic_until_fixed ... FAILED
...
failures:

---- sp_borsh::srlabs::trigger_decode_panic_until_fixed stdout ----

thread 'sp_borsh::srlabs::trigger_decode_panic_until_fixed' (552077) panicked at
substrate/primitives/src/sp_borsh/bitvec.rs:102:1:
Reader::read errored but didn't set the error
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Risk

An attacker can submit well-formed SCALE input over the network that passes all read operations but fails Borsh validation, causing a panic during decoding. When decoding is applied to untrusted data, such as extrinsics, messages, or persisted state derived from user input, this panic can crash the process or abort execution. In the Serai DEX context, this can result in denial of service of a node, and may have consensus or availability impact depending on where decoding occurs.

Mitigation

Ensure that Borsh deserialization errors are propagated as normal SCALE decode errors instead of assuming a prior read failure. The adapter must handle Borsh errors independently of SCALE input errors and must never unwrap an absent stored error.

8.2 S3-729: serai-primitives: Panic on invalid public key during muSig key aggregation

Attack scenario	**An attacker wants to perform DoS attacks to halt or slow down block production**
Classification	VLN-7: Reachable Panic
Component	primitives crate
Tracking	https://github.com/serai-dex/serai/issues/729
Attack impact	T-7: An attacker is able to arbitrarily panic the runtime by calling an extrinsic
Severity	High
Status	Mitigated

Background

The Serai primitives define base level features and relationships between objects for the DEX.

Issue description

In serai-primitives, the `ExternalValidatorSet::musig_key` function decompresses and aggregates validator public keys using Ristretto operations. It does not validate the input keys before attempting decompression. If any key fails to decompress as a valid Ristretto point, the code calls `.expect("invalid participant")`, which causes an immediate runtime panic. Furthermore, even after decompression, the function unwraps the result of `dkg::musig_key(...)` without handling errors, introducing a second potential panic. Both branches expose the system to unguarded crashes if any participant supplies malformed or adversarial key material.

Affected code from `src/validator_sets/mod.rs` that even mentions this problem:

```
  /// The MuSig public key for a validator set.
  ///
  /// This function panics on invalid points as keys and on invalid input, per
the definition of
  /// `dkg::musig::musig_key`.
  pub fn musig_key(&self, keys: &[Public]) -> Public {
    let mut decompressed_keys = vec![];
    for key in keys {
      decompressed_keys.push(
        <Ristretto as GroupIo>::read_G::<&[u8]>(&mut key.0.as_slice())
          .expect("invalid participant"),
      );
    }
    dkg::musig_key::<Ristretto>(self.musig_context(),
&decompressed_keys).unwrap().to_bytes().into()
  }
```

This function is currently only called in `serai-validator-sets` - which is not ready to be audited yet - in `src/lib.rs`:

```
  #[pallet::validate_unsigned]
  impl<T: Config> ValidateUnsigned for Pallet<T> {
    type Call = Call<T>;
    fn validate_unsigned(_: TransactionSource, call: &Self::Call) ->
TransactionValidity {
      // Match to be exhaustive
      match call {
        Call::set_keys { network, ref key_pair, ref signature_participants, ref
signature } => {
```

```
    [...]

        // Verify the signature with the MuSig key of the signers
        match signature {
          Signature::Ristretto(signature) => {
            if !set
              .musig_key(&signers)
              .verify(&set.set_keys_message(key_pair), &signature.0.into())
```

As `set_keys` is an unsigned call anybody can submit this with malformed keys (with some requires needed to be fulfilled) which would result in a crash when processed.

Risk

An attacker can fake or modify a valid `set_keys` call that includes an invalid or malformed public key (e.g., a 32-byte sequence not representing a valid Ristretto point). During key verification, the application will panic on decompressing this invalid key, crashing the node or invalidating the block. This creates a critical Denial of Service vector.

Mitigation

Validate all public keys before attempting decompression. Reject any invalid keys with recoverable errors rather than panicking. Avoid unwrap or expect in consensus logic and propagate errors through proper Result types. Ensure the aggregation function only operates on fully validated and decompressed inputs.

8.3 S3-741: Pre-economic-security liquidity addition enables protocol-owned liquidity extraction

Attack scenario	**An attacker does extract protocol-owned liquidity**
Classification	VLN-15: Insecure Business Logic
Component	dex-pallet, coins-pallet
Tracking	https://github.com/serai-dex/serai/issues/741
Attack impact	T-4: Exploit a vulnerability in Serai resulting in stolen liquidity
Severity	High
Status	Mitigated

Background

The DEX pallet manages liquidity pools for SRI-XYZ pairs and provides functionality for adding/removing liquidity and swapping tokens. The `SwapAndAddLiquidity` instruction allows users to swap external coins to SRI and add both as liquidity to pools, while `SwapToStakedSri` is a pre-economic-security instruction that mints new SRI and stakes it in exchange for external assets that become protocol-owned liquidity.

Issue description

During pre-economic security, an attacker can exploit the asymmetry between liquidity addition and removal to extract protocol-owned liquidity:

1) Attacker calls `SwapAndAddLiquidity` to obtain liquidity tokens representing a share of the pool.
2) Later, `SwapToStakedSri` is executed, which adds protocol-owned external reserves to the pool.
3) Attacker immediately calls `remove_liquidity`. Since pre-economic security burns the SRI portion and returns only external assets, the attacker captures a larger share of external assets than originally deposited.

The root cause is that `add_liquidity` in the DEX pallet currently lacks any pre-economic-security guard, allowing liquidity additions when they should be prohibited until global economic security is achieved.

Risk

This vulnerability allows attackers to extract protocol-owned liquidity, undermining the economic security bootstrapping mechanism. The impact is proportional to the amount of protocol-owned liquidity added via `SwapToStakedSri`, potentially draining significant external assets from the system before economic security is achieved.

Mitigation

The planned mitigation is to have the `EconomicSecurity` pallet implement the `AllowMint` trait to limit liquidity token mints to when capacity exists. This would prevent `add_liquidity` from succeeding during pre-economic-security.

8.4 S3-753: `set_keys` validation vs dispatch mismatch

Attack scenario	**An attacker triggers a reachable panic via a session mismatch, causing a denial of service**
Classification	VLN-7: Reachable Panic
Component	validator-sets pallet
Tracking	https://github.com/serai-dex/serai/issues/753
Attack impact	T7: Perform DoS attacks to halt or slow down block production
Severity	High
Status	Mitigated

Background

External-network messages are submitted to Serai as Batches and must be authenticated by a threshold signature from the relevant external validator set. The on-chain public key used to verify those signatures is the set's oraclization key. When a new validator set is decided, that set publishes its new public keys on-chain via the unsigned `set_keys` call (oraclization key, plus the external-network key), so the runtime can validate subsequent Batch signatures and other set-authenticated messages.

Issue description

There exists a mismatch between the validator set session validated in `validate_unsigned` and the one mutated inside the `set_keys` extrinsic. The `validate_unsigned` logic validates the extrinsic signature using validator keys from the latest_decided_session. However, inside `set_keys` the keys for the current_session are then set. As a result, when `current_session != latest_decided_session`, which is the case during pending-handover between validator sets, keys are written for a different set than the one whose signature was checked.

Risk

Upon network bootstrap, `latest_decided_session` will be set for Session 0, but `current_session` will be None. Due to this issue, the `set_keys` extrinsic will therefore panic due to `Self::current_session(NetworkId::from(network))` being `None`. As a result, batch publication for an external network will never start since keys can never be stored successfully.

Mitigation

Ensure that the `set_keys` extrinsic sets the keys for `latest_decided_session`.

8.5 S3-758: Inverted GRANDPA equivocation proof validation allows bypass and false slashing

Attack scenario	**An attacker wants to control block production and consensus**
Classification	VLN-13: Incorrect Slashing Logic
Component	runtime
Tracking	https://github.com/serai-dex/serai/issues/758
Attack impact	T-6: Control block production and consensus
Severity	High
Status	Mitigated

Background

Equivocation proofs are a feature for BABE and GRANDPA to report maliciously behaving validators so they can be slashed for the behavior. A validator needs to verify such reports to ensure that they are valid.

Issue description

The function `check_equivocation_proof` in substrate/runtime/src/wasm/babe_grandpa/grandpa.rs decodes a SCALE-encoded `EquivocationProof` and calls `sp_consensus_grandpa::check_equivocation_proof(equivocation_proof)`. The current implementation handles the return value with inverted semantics.

`sp_consensus_grandpa::check_equivocation_proof` returns `true` when the proof is valid. In this case though `check_equivocation_proof` returns `None` and returns the offender when the function returns `false`, effectively treating an invalid proof as valid:

```
if sp_consensus_grandpa::check_equivocation_proof(equivocation_proof) {
  None?;
}
```

The runtime integrates this function into the slash-reason verification path inside `apply_extrinsic` before `Executive::apply_extrinsic` executes the extrinsic. The return value directly determines whether the runtime accepts an unsigned slash report.

On a side-note, the same check in BABE is correctly implemented.

Risk

When a validator produces a valid equivocation proof, the runtime rejects it at this layer. Malicious validators can therefore commit GRANDPA equivocations and avoid slashing, which weakens finality guarantees.

At the same time, the inverted logic opens the door to false positives. An attacker can construct an invalid equivocation proof that fails the underlying `sp_consensus_grandpa::check_equivocation_proof` call but still passes the runtime check. If the remaining succeed, the runtime will accept the unsigned slash report, resulting in a direct griefing vector where an attacker can attempt to trigger unjustified slashing against honest validators, causing financial damage, network instability and loosing trust in Serai.

Mitigation

We recommend correcting the boolean handling so that the runtime treats a true return value from `sp_consensus_grandpa::check_equivocation_proof` as a valid proof and a false value as invalid (as it is the case in the BABE implementation):

```
if !sp_consensus_grandpa::check_equivocation_proof(equivocation_proof) {
  None?;
}
```

We also recommend introducing explicit unit and integration tests that assert the expected behavior for both valid and invalid GRANDPA equivocation proofs at the runtime boundary.

8.6 S3-759: Extra `floor(chunk / N)` transfer on resolve block can overdraw pending liquidity and trigger panic

Attack scenario	**An attacker does trigger a panic in pre_inherents and disrupt block production**
Classification	VLN-7: Reachable Panic
Component	in-instructions pallet
Tracking	https://github.com/serai-dex/serai/issues/759
Attack impact	T-7: Perform DoS attacks to halt or slow down block production
Severity	High
Status	Mitigated

Background

The In-Instructions pallet executes inbound instructions from external networks and manages how incoming assets are integrated into Serai.

Issue description

The `pre_inherents()` hook inside the in-instructions pallet, drip-feeds already-queued pending liquidity into DEX pools over a fixed number of blocks instead of adding the full amount in a single block. This mechanism works on **chunks** of external liquidity. Each chunk is distributed over `N` blocks to smooth liquidity inflow and avoid sudden pool changes.

The intended behavior is the following:

If a chunk with amount `chunk` is registered at block `B`, then the liquidity is added incrementally:

- for blocks `B + 1` through `B + N`, the pool receives `floor(chunk / N)` each block
- then finalize the chunk by applying the rounding remainder `remainder = chunk - floor(chunk / N) * N`, so the total transferred for that chunk is exactly `chunk`

During finalization, the code computes the remainder:

```
let remainder = (pending_liquidity_resolving_this_block -
  resolving_pending_liquidity_added_per_block)
  .expect("amounts added over time exceeded total");
```

But on that same block, the transferred amount is built as:

```
pending_liquidity_per_block = (pending_liquidity_per_block + remainder)
  .expect("pending liquidity exceed supply (bound to representable in
`amount`)");

// ...

Coins::<T>::transfer_fn(
  serai_abi::in_instructions::pending_liquidity_address(coin),
  serai_abi::dex::address(coin),
  Balance { coin: coin.into(), amount: pending_liquidity_per_block },
)
```

So the resolving chunk effectively contributes `floor(chunk / N)` once more on finalization, yielding:

```
floor(chunk / N) * (N + 1) + remainder
```

instead of the intended:

```
floor(chunk / N) * N + remainder
```

This leads to **one extra `floor(chunk / N)` transfer** for the resolving chunk.

This can cause an overdraw attempt from the pending liquidity address and can trigger a `panic`.

The panic may be temporarily avoided if newer chunks still provide enough balance to cover the extra transfer. However, once those chunks finish dripping and the buffer disappears, the panic path becomes reachable.

Risk

As logic runs in `pre_inherents`, such a panic can interrupt block processing and therefore represents a **chain liveness risk**. Even when panic is avoided, chunk level accounting becomes incorrect.

Mitigation

Ensure that a resolving chunk does not contribute an additional `floor(chunk / N)` on the same block it is finalized.

The resolve block transfer should be computed using the **post resolve rate**, plus the explicit remainder for the resolving chunk. In code:

```
{
  let new_pending_liquidity_per_block = (pending_liquidity_per_block -
    pending_liquidity_per_block_from_resolving)
    .expect("more pending liquidity per block from this block than from all
blocks");
  PendingLiquidityPerBlock::<T>::set(
    coin,
    Some(new_pending_liquidity_per_block).filter(|value| *value != Amount(0)),
  );
  // Add the following line
  pending_liquidity_per_block = new_pending_liquidity_per_block;
}
```

8.7 S3-760: Inconsistent rounding between queue and resolve can cause panic

Attack scenario	**An attacker does trigger a panic in pre_inherents and disrupt block production**
Classification	VLN-7: Reachable Panic
Component	in-instructions pallet
Tracking	https://github.com/serai-dex/serai/issues/760
Attack impact	T-7: Perform DoS attacks to halt or slow down block production
Severity	High
Status	Mitigated

Background

The In-Instructions pallet executes inbound instructions from external networks and manages how incoming assets are integrated into Serai.

Issue description

`swap_to_staked_sri()` queues the amount (chunk) to be added to the pool, and `pre_inherents()` drip feeds that amount into the pool over `N` blocks, resolving the chunk once the full amount has been distributed.

The code mixes two different rounding approaches for **queuing** versus **resolving**.

Update of rate during queue time (link)

```
*pending_liquidity_per_block = Some((
        existing +
        Amount(external_balance_in
            .amount
            .0
            .checked_div(u64::from(BLOCKS_TO_ADD_PENDING_LIQUIDITY_OVER))
            .unwrap_or(0)))
    .expect(
        "pending liquidity per block exceeded supply (bound to be representable
in `Amount`)",
    ));
```

Since multiple same coin chunks can be queued in a single block, this code is executed for each chunk. The abstracted formula for the rate update is therefore:

```
sum_i floor(a_i / N)
```

Removal of rate during resolve (link)

```
let pending_liquidity_per_block_from_resolving = Amount(
    pending_liquidity_resolving_this_block
    .0
    .checked_div(u64::from(BLOCKS_TO_ADD_PENDING_LIQUIDITY_OVER))
    .unwrap_or(0),
);
```

Here, `pending_liquidity_resolving_this_block` already contains the aggregation of all same coin chunks resolving in that block. The abstracted formula for the rate update on removal is therefore:

```
floor((sum_i a_i) / N)
```

In general, these two expressions are not equal:

```
floor((sum_i a_i) / N) >= sum_i floor(a_i / N)
```

As a result, the resolve step can attempt to remove more rate than was ever added for that block. This could trigger a panic.

More precisely, the panic becomes reachable when:

`pending_liquidity_per_block < floor((sum_i a_i) / N)` at resolve time.

Similar to Issue #769, this panic may temporarily be absorbed by other chunks that still contribute to `pending_liquidity_per_block`. However, once those chunks resolve as well, the panic becomes reachable.

Risk

As logic runs in `pre_inherents`, such a panic can interrupt block processing and therefore represents a **chain liveness risk**.

Mitigation

Use one rounding model for both queue and resolve.

8.8 S2-733: serai-abi: Inverted nonce logic in transaction validation

Attack scenario	**An attacker abuses inverted nonce checks to clog the transaction pool and block valid transactions**
Classification	VLN-9: Storage Exhaustion
Component	abi crate
Tracking	https://github.com/serai-dex/serai/issues/733
Attack impact	T7: Hacker wants to perform DoS attacks to halt or slow down block production
Severity	Medium
Status	Mitigated

Background

Serai uses nonces to sequence signed transactions on a per-account basis. Each signed transaction includes the signer's nonce, and the runtime tracks the next expected nonce for that account.

This mechanism serves two purposes: First, it prevents replay of already-consumed signed transactions. Second, it enforces per-account transaction ordering. A transaction with the current expected nonce can be processed immediately, while transactions with higher nonces may be kept in the pool until earlier transactions from the same account are accepted.

Issue description

In serai-abi, the validate_except_fee function is called to validate both signed and unsigned transactions. For signed transactions, the nonce validation logic is incorrect.

Specifically, the comparison between the account's on-chain next_nonce and the transaction's nonce is inverted:

```
          match next_nonce.cmp(nonce) {
            core::cmp::Ordering::Less => {
              Err(TransactionValidityError::Invalid(InvalidTransaction::Stale))?
            }
            core::cmp::Ordering::Equal => {}
            // Because this is greater than the on-chain nonce, add a
requirement of the prior
            // nonce
            core::cmp::Ordering::Greater => requires.push(tag_for_nonce(*nonce -
1)),
          }
```

As a result:

- Transactions with a nonce **greater than** the on-chain nonce are incorrectly treated as stale.
- Transactions with a nonce **less than** the on-chain nonce incorrectly introduce a dependency on a prior nonce.

Risk

Due to this issue stale transactions may be incorrectly accepted into the pool and remain pending until they are eventually evicted due to longevity limits, increasing pool pressure and wasting validation resources.

Furthermore, valid transactions that use a higher nonce than the current on-chain value are systematically rejected from the pool. This can prevent users from submitting follow-up transactions,

effectively causing account-level transaction denial until the pool state is manually or naturally cleared.

Mitigation

Fix the comparison to be `nonce.cmp(next_nonce)`, which will make the logic of the match statement correct.

8.9 S2-744: Unbounded State Growth via Zero Fee Dust Accounts

Attack scenario	**An attacker creates large amounts of dust accounts and bloats Serai state cheaply.**
Classification	VLN-9: Storage Exhaustion
Component	coins pallet
Tracking	https://github.com/serai-dex/serai/issues/744
Attack impact	T-7: Perform DoS attacks to halt or slow down block production
Severity	Medium
Status	Acknowledged

Background

The coins pallet is responsible for maintaining token balances and handling transfers between accounts. The `transfer` extrinsic allows users to send arbitrary non-zero amounts of SRI to any valid address, creating a new account entry if the recipient does not yet exist.

Issue description

Serai does not implement an existential deposit. The coins pallet permits the creation of accounts with any non zero balance (link). Consequently, sending 1 atomic unit of SRI to a fresh address creates a dust account and inserts a storage entry into `Balances<T, I>`.

The estimated storage footprint per such account is approximately:

- **SeraiAddress key**: ~32 bytes
- **Coin key**: ~1-3 bytes (discriminator + variant)
- **Blake2_128Concat prefix**: 16 bytes
- **Amount value**: 8 bytes (u64)
- **Storage overhead**: ~32-50 bytes (Merkle Patricia, metadata)

Total per account: roughly 100 bytes of on chain storage.

Transaction fees are designed to be weight based (link). However, the current implementation defaults transaction weight to zero (link). Signed transactions include a user-specified fee field (link), and the runtime only checks balance sufficiency when charging it (link). There is no positive minimum fee check, so a fee of zero is valid.

A signed transaction can batch up to 8 signed calls (link).

The serialized size of one `coins::transfer` call is about 43 bytes.

- 1 byte for outer `Call::Coins` (link)
- 1 byte for `coins::Call::transfer` (link)
- 32 bytes for `SeraiAddress` (link)
- 9 bytes for `Balance` (`Coin` 1 byte + `Amount(u64)` 8 bytes) (link, link).

The fixed overhead of one signed transaction is about 144 bytes. It includes:

- 1-byte signed-calls length prefix (link)
- `ExplicitContext` of 78 bytes when `include_by=None` (link)
- 65-byte signature encoding (1-byte enum tag + 64-byte Ristretto signature) (link, link).

So an 8-call dusting transaction is approximately:

```
144 + 8 * 43 = 488 bytes
```

Block size is capped at 512 KiB (link). The runtime pre-accounts 149 bytes per block (`Header::SIZE` + 4) (link link). That leaves 524,139 bytes for transactions. At ~488 bytes each, about 1,074 such signed transactions fit per block. Each carries 8 transfers, so about 8,592 new dust accounts per block.

Using the ~100 bytes/account assumption, 100 GB of added state corresponds to about 1,000,000,000 dust accounts. At ~8,592 accounts per block, this is about 116,387 blocks. With a 6-second target block time (link), this is about 8.1 days of sustained spam.

Risk

An attacker can continuously create dust accounts at zero cost, leading to unbounded state growth. Over time, this increases node storage requirements, impacts state synchronization, raises hardware costs for validators, and may degrade network performance.

Because no existential deposit and no positive minimum fee are enforced, the attack is economically unconstrained except by bandwidth and block limits. Sustained spam over several days can add tens or hundreds of gigabytes of permanent state.

Mitigation

Introduce an existential deposit to prevent the creation of accounts below a minimum balance threshold.

Enforce non-zero minimum transaction fees and ensure weight-based fee calculation is properly implemented.

Optionally, implement storage rent, account reaping, or state growth limits to bound long-term storage expansion.

8.10 S2-746: `quote_for_out` function in dex module may panic

Attack scenario	**An attacker crafts an extrinsic that triggers a runtime panic path, causing execution traps that can be repeated to deny service to the chain.**
Classification	VLN-7: Reachable Panic
Component	dex pallet
Tracking	https://github.com/serai-dex/serai/issues/746
Attack impact	T7: Perform DoS attacks to halt or slow down block production.
Severity	Medium
Status	Mitigated

Background

The DEX pallet manages liquidity pools and token swaps between SRI and supported external assets using constant-product pricing. The `swap_for` extrinsic computes the required input for a user-specified exact output amount, and it relies on `quote_for_out` for that calculation.

Issue description

The quote_for_out function may panic in case `out_reserve == amount_out.0` resulting in `proposed_out_reserve` to be `0` and `current_k.div_ceil(u128::from(proposed_out_reserve));` to panic due to an attempted division by `0`.

The following test case, which was added to `substrate/primitives/src/dex/mod.rs`, triggers the panic:

```
#[test]
#[should_panic]
fn quote_for_out_full_reserve_panics() {
  let reserves = Reserves { sri: Amount(100_000), external_coin: Amount(10_000)
};
  let sri_in = Premise::establish(Coin::Serai,
Coin::from(ExternalCoin::Bitcoin)).unwrap();
  let _ = sri_in.quote_for_out(reserves, Amount(10_000));
}
```

Risk

Since the `quote_for_out` function is used by the swap_for extrinsic, any user who can determine the current `out_reserve` can deterministically trigger this panic, causing a runtime trap and degrading Serai chain availability. If enough malicious extrensic calls exist in the mempool, block producers might miss their slot.

Mitigation

Inside `quote_for_out`, ensure that `proposed_out_reserve` is not zero before calling `div_ceil`. Furthermore, ensure that the module covers all edge cases instead of relying exclusively on randomized values as done in the quote_does_not_panic test.

8.11 S2-761: Zero drip rate causes pending liquidity chunks to remain unresolved

Attack scenario	**An attacker does delay or prevent pending liquidity from reaching the DEX pool**
Classification	VLN-15: Insecure Business Logic
Component	in-instructions pallet
Tracking	https://github.com/serai-dex/serai/issues/761
Attack impact	T-7: Perform DoS attacks to halt or slow down block production
Severity	Medium
Status	Mitigated

Background

The In-Instructions pallet executes inbound instructions from external networks and manages how incoming assets are integrated into Serai.

Issue description

The `pre_inherents()` hook inside the in-instructions pallet drip-feeds already queued pending liquidity into DEX pools over a fixed number of blocks instead of adding the full amount in a single block. This mechanism works on **chunks** of external liquidity. Each chunk is distributed over `N` blocks to smooth liquidity inflow and avoid sudden pool changes.

`PendingLiquidityPerBlock` is used as both:

- the numeric drip rate, and
- the guard that checks whether, for the given external coin, something has to be processed.

That breaks when the true drip rate is zero but unresolved chunks still exist.

Let's look at the following scenario:

Assume `N` is large, and two tiny same-coin chunks are queued:

- Chunk `A` queued at block `B`, with amount `< N`
- Chunk `B` queued at block `B + 1`, with amount `< N`

So the state after queueing is:

- `PendingLiquidity[coin, B] = A`
- `PendingLiquidity[coin, B+1] = B`
- `PendingLiquidityPerBlock[coin] = Some(0)` (because both floor-divide to 0)

At resolve time for block `B`:

1. Resolves `A` by taking the historical entry.
2. Computes `pending_liquidity_per_block_from_resolving`, which is zero when the chunk `< N`.

```
let pending_liquidity_per_block_from_resolving = Amount(
    pending_liquidity_resolving_this_block
        .0
        .checked_div(u64::from(BLOCKS_TO_ADD_PENDING_LIQUIDITY_OVER))
        .unwrap_or(0),
);
```

3. Computes the new rate, which is again zero (`0 - 0 = 0`).

```
let new_pending_liquidity_per_block = (pending_liquidity_per_block -
    pending_liquidity_per_block_from_resolving)
.expect("more pending liquidity per block from this block than from all
blocks");
```

4. Stores `None` when the rate is zero.

```
PendingLiquidityPerBlock::<T>::set(
    coin,
    Some(new_pending_liquidity_per_block).filter(|value| *value != Amount(0)),
);
```

5. Transfers the remainder for resolving chunk A (which is, by definition, the whole chunk A).

Now the next block should resolve `PendingLiquidity[coin, B+1]` (chunk B), but `PendingLiquidityPerBlock[coin]` is now `None` and the function exits early:

```
let Some(mut pending_liquidity_per_block) =
PendingLiquidityPerBlock::<T>::get(coin) else {
    continue;
};
```

This has the effect that chunk `B` is never resolved unless a new later queue repopulates `PendingLiquidityPerBlock`. As a result, the funds for the unresolved chunk remain in the pending-liquidity address and the pool does not receive them on schedule. In theory, resolution may be delayed indefinitely if no new chunks arrive for that coin.

Risk

Liquidity can become permanently stuck in the pending-liquidity address. This results in funds not reaching the DEX pool, leading to incorrect pool liquidity.

Mitigation

Do not use `PendingLiquidityPerBlock` as the guard for whether pending liquidity needs to be processed. Resolution should instead check whether historical `PendingLiquidity` entries exist for the current block.

8.12 S2-763: SwapOut Allows Outbound Actions On Halted Destination Networks

Attack scenario	**An attacker does bypass halt semantics to create new outbound obligations on a halted network**
Classification	VLN-15: Insecure Business Logic
Component	in-instructions pallet, coins pallet, signals pallet
Tracking	https://github.com/serai-dex/serai/issues/763
Attack impact	T-4: Exploit a vulnerability in Serai resulting in stolen liquidity
Severity	Medium
Status	Mitigated

Background

The Signals pallet provides a mechanism to permanently halt an external network. Its own documentation explicitly states that halting a network only records the halt state, and that downstream consumers must use this information to prevent further actions from occurring.

Issue description

The `validate_execute_batch()` check inside the in-instructions pallet only rejects a batch when the batch's own network is halted.

```
let network = batch.batch.network();

// Verify the network isn't halted
if serai_signals_pallet::Pallet::<T>::halted(network) {
    Err(InvalidTransaction::Custom(0))?;
}
```

That network is the network the batch is coming from, as defined in `Batch`.

This is sufficient to stop new inbound batches originating from a halted network. It is not sufficient to stop creating new outbound obligations on a halted network.

The gap is caused by `SwapOut`. `SwapOut` is defined in terms of a `minimum_to_receive: ExternalBalance`, meaning the output coin is chosen independently of the batch's source network.

During execution, `SwapOut`:

```
InInstruction::SwapOut { instruction, minimum_to_receive } => {
    serai_dex_pallet::Pallet::<T>::swap(
            RawOrigin::Signed(in_instructions_address).into(),
            external_balance_in.into(),
            minimum_to_receive.into(),
            )?;

    let coin = minimum_to_receive.coin;
    let received_amount = Coins::<T>::balance(in_instructions_address,
Coin::from(coin));
    let received = ExternalBalance { coin, amount: received_amount };
    // This is disallowed as this must only be called with `CoinsInstance`,
which this does
    #[expect(clippy::disallowed_methods)]
    Coins::<T>::burn_with_instruction(
            RawOrigin::Signed(in_instructions_address).into(),
            OutInstructionWithBalance { instruction, balance: received },
            )?;
}
```

1. swaps the inbound balance into `minimum_to_receive.coin`, and then
2. immediately calls `burn_with_instruction()` on that received external coin.

So while batch validation only checks whether the **source** network is halted, `SwapOut` may still emit a burn for a different, already halted **destination** network.

For example, assume:

- Ethereum is live
- Bitcoin is halted

A valid Ethereum batch can still contain a `SwapOut` whose `minimum_to_receive.coin` is Bitcoin. The halt check passes because the batch came from Ethereum, not Bitcoin. Execution then swaps the Ethereum-side inbound balance into Bitcoin and emits a Bitcoin `BurnWithInstruction`.

That burn is then propagated downstream and results in an actual transfer on the external halted network.

Risk

This circumvents the protocol's emergency stop semantics.

If a network is halted because its outbound path is unsafe, Serai should not create any new outbound obligations on that network. Under the current implementation, a non-halted source network can still be used to create new outbound work for a halted destination network through `SwapOut`.

This means the halt mechanism can be bypassed in cross-network flows, undermining incident response and allowing further processing on a network that operators explicitly intended to freeze.

Mitigation

Enforce halt checks on the **destination** network as well, not only on the batch's source network.

At minimum, `SwapOut` should reject execution when `minimum_to_receive.coin.network()` is halted.

8.13 S2-771: Expired external-network support can still be counted toward signal thresholds

Attack scenario	**An attacker uses stale cached support to finalize a halt or retirement**
Classification	VLN-15: Insecure Business Logic
Component	signals pallet
Tracking	https://github.com/serai-dex/serai/issues/771
Attack impact	T-7: perform DoS attacks to halt or slow down block production
Severity	Medium
Status	Mitigated

Background

The signals pallet lets validators express protocol-level intent, specifically to retire the Serai protocol in favor of a new one or to permanently halt an external network.

Issue description

The signals pallet uses `ValidatorSetsInFavor` as a cached per-validator-set support flag, while the actual live votes are stored in `Favors`.

A favor expires based on the Serai session, as each `favor(...)` stores `favor_until_serai_session = current_serai_session + 1`. Live support is recomputed in `tally_for_network`, which ignores expired `Favors` and then updates `ValidatorSetsInFavor` accordingly.

The issue is that `tally_for_all_networks` does not recompute support from `Favors`. It only checks whether `ValidatorSetsInFavor` contains an entry for each network's current validator set.

As a result, if an external network was previously tallied as "in favor", and its underlying favors later expire due to Serai session advancement, that external network will still count toward the global threshold until something explicitly retallies that same network and clears the cache.

This is reachable because external validator-set sessions do not automatically advance with Serai sessions. They advance on external handover in validator-sets, triggered from the external batch path in in-instructions. Therefore, an external network can keep the same cached `ValidatorSetsInFavor` entry across multiple Serai sessions, even after all underlying favors are expired.

Risk

Global threshold checks for **Halt** and **Retire** can succeed using expired support.

This has two concrete consequences:

1. A network halt can be finalized using support that is no longer live.
2. A retirement signal can be locked in using support that is no longer live.

That violates the intended lifetime semantics of `Favors` and allows protocol-critical decisions to be made from stale state.

Mitigation

In our opinion `ValidatorSetsInFavor` should not be used as the source of truth for global threshold checks.

Instead we recommend either option:

- Recompute each network's support from live `Favors` inside `tally_for_all_networks`; or

- invalidate/recompute all `ValidatorSetsInFavor` entries whenever the Serai session advances.

8.14 S1-742: build.rs artifact selection is non-deterministic

Attack scenario	**An attacker exploits non-deterministic WASM artifact selection to introduce inconsistent runtimes across nodes.**
Classification	VLN-11: Outdated Crates
Component	build.rs
Tracking	https://github.com/serai-dex/serai/issues/742
Attack impact	T-10: Hacker wants to perform a supply chain attack to compromise the Serai network.
Severity	Low
Status	Mitigated

Background

Serai uses a custom build script to make builds less opaque and reproducible ....

Issue description

To locate the produced WASM runtime artifact, the custom `build.rs` recursively scans the `target` directory and picks the last filename matching the `<pkg_name>.wasm` format (link). This approach is non deterministic as stale build artifacts could be picked. Furthermore, `fs::read_dir` order can change between calls, causing different results over multiple runs.

Risk

This issue could result in different runtimes to be run by Serai nodes, potentially causing behavioral differences.

Mitigation

Determine the exact output path of the produced WASM runtime. We would recommend the second approach mentioned in the comments above the code. Pass `--message-format=json` to the build command and then parse the output, even if this would require a JSON parser build dependency.

Example code of how to deterministically find the WASM artifact:

```
let file = cargo_env("CARGO_PKG_NAME").replace('-', "_") + ".wasm";
let mut artifacts = HashSet::<PathBuf>::new();
for line in stdout.lines() {
  let line = line.expect("failed to read line from `cargo` JSON stdout");
  let Ok(json) = serde_json::from_str::<serde_json::Value>(&line) else {
    continue;
  };

  if json.get("reason").and_then(|r| r.as_str()) != Some("compiler-artifact") {
    continue;
  }

  let Some(filenames) = json.get("filenames").and_then(|f| f.as_array()) else {
    continue;
  };
  for filename in filenames {
    let Some(filename) = filename.as_str() else { continue };
    let path = PathBuf::from(filename);
    if path.file_name().and_then(|n| n.to_str()) == Some(file.as_str()) {
      artifacts.insert(path);
    }
  }
```

```
}

assert!(!artifacts.is_empty(), "nested build succeeded but did not emit a
`{file}` artifact");
assert_eq!(
  artifacts.len(),
  1,
  "nested build emitted multiple `{file}` artifacts: {artifacts:?}"
);
let artifact = artifacts.into_iter().next().unwrap();
```

8.15 S1-745: `decrease_allocation` function in validator-sets sessions module may panic

Attack scenario	**An attacker crafts an extrinsic that triggers a runtime panic path, causing execution traps that can be repeated to deny service to the chain.**
Classification	VLN-7: Reachable Panic
Component	validator-sets pallet
Tracking	https://github.com/serai-dex/serai/issues/745
Attack impact	T7: perform DoS attacks to halt or slow down block production.
Severity	Low
Status	Mitigated

Background

The validator-sets pallet manages validator participation and stake allocation across networks and sessions. The `deallocate` extrinsic reduces a validator's allocated stake and updates session-level accounting used for economic security checks.

Issue description

The decrease_allocation function inside the validator-sets sessions module may panic when a validator that is part of a set tries to deallocate more stake than the total amount staked by this set. The panic is triggered by the function's usage of `expect` on the result of a checked subtraction that may return `None` in case of underflow. This issue can be triggered through the deallocate extrinsic defined inside the `validator-sets` pallet. This affects validators in the latest decided external set.

The following test demonstrates the issue by triggering the second `expect` (a deallocation of 1 is enough in this case since the validator set has staked 0):

```
#[cfg(test)]
mod tests {
  use super::*;
  use serai_abi::{
    Call as AbiCall,
    primitives::{
      network_id::{ExternalNetworkId, NetworkId},
      balance::Amount,
    },
    validator_sets,
  };

  fn panic_message(payload: Box<dyn core::any::Any + Send>) -> String {
    if let Some(msg) = payload.downcast_ref::<String>() {
      return msg.clone();
    }
    if let Some(msg) = payload.downcast_ref::<&str>() {
      return (*msg).to_owned();
    }
    "non-string panic payload".to_owned()
  }

  #[test]
  fn runtimecall_validator_sets_deallocate_can_panic() {
    let accounts: Vec<AccountId> =
      (0..5).map(|i| serai_abi::primitives::crypto::Public([i as u8;
32]).into()).collect();
    let genesis = generate_genesis(&accounts);
```

```
    let panic = std::panic::catch_unwind(|| {
      BasicExternalities::execute_with_storage(&mut genesis.clone(), || {
        initialize_block(1);

        let call = AbiCall::ValidatorSets(validator_sets::Call::deallocate {
          network: NetworkId::External(ExternalNetworkId::Ethereum),
          amount: Amount(1),
        });
        let call = RuntimeCall::from(call);
        assert!(matches!(call, RuntimeCall::ValidatorSets(_)));

        let tx_len = call.encoded_size();
        let tx_hash = sp_core::blake2_256(&call.encode());
        serai_runtime::Core::start_transaction(tx_len);
        let _ =
call.dispatch(serai_runtime::RuntimeOrigin::signed(accounts[0].clone()));
        serai_runtime::Core::end_transaction(tx_hash);
      });
    })
    .expect_err("expected panic for oversized deallocation");

    let msg = panic_message(panic);
    assert!(
      msg.contains("validator in set deallocated stake the set didn't have"),
      "unexpected panic message: {msg}"
    );
  }
}
```

Risk

A validator in a selected set can trigger a runtime panic via deallocate, causing execution traps and potentially availability degradation (DoS impact) for block production.

Mitigation

Do not make use of `expect` and properly handle the subtraction underflow by propagating e.g. a `DeallocationError::NotEnoughAllocated` error.

8.16 S1-747: Runtime panic due to multiplication overflow inside `sri_value` function

Attack scenario	**An attacker crafts an extrinsic that triggers a runtime panic path, causing execution traps that can be repeated to deny service to the chain**
Classification	VLN-7: Reachable Panic
Component	economic-security pallet
Tracking	https://github.com/serai-dex/serai/issues/747
Attack impact	T7: Perform DoS attacks to halt or slow down block production.
Severity	Low
Status	Mitigated

Background

The Economic Security pallet checks whether Serai has enough SRI stake to safely back each external network. It retrieves the current pool reserves and derives a price quote from them, then uses these prices to compute the required stake, determines when a network reaches economic security. Furthermore, the pallet exposes checks that allow other pallets to evaluate whether to permit or restrict minting and liquidity operations.

Issue description

The sri_value function can panic when an external/SRI pool is extremely imbalanced such that the external reserve is very small relative to the SRI reserve. In that case, `quote_per_unit` inside the `sri_value` function becomes very big and the multiplication `u128::from(balance.amount.0) * quote_per_unit)` may overflow.

Following is a test, added to `substrate/economic-security/src/tests.rs`, that triggers the overflow:

```
#[test]
#[should_panic(expected = "attempt to multiply with overflow")]
fn economic_security_oracle_mul_overflow_panics() {
  new_test_ext().execute_with(|| {
    Core::start_transaction(0);

    // Initialize the Monero pool so `sri_quote` is available.
    LiquidityTokens::mint(
      SeraiAddress([0; 32]),
      Balance { coin: ExternalCoin::Monero.into(), amount: Amount(1) },
    )
    .unwrap();

    // Make quote_per_unit extremely large
    let pool = serai_abi::dex::address(ExternalCoin::Monero);
    Coins::mint(pool, Balance { coin: ExternalCoin::Monero.into(), amount:
Amount(1) }).unwrap();
    Coins::mint(pool, Balance { coin: Coin::Serai, amount: Amount(u64::MAX)
}).unwrap();

    // Trigger overflow
    let _ = EconomicSecurity::sri_value(ExternalBalance {
      coin: ExternalCoin::Monero,
      amount: Amount(u64::MAX),
    });
  });
}
```

Risk

If a pool's SRI/external ratio becomes sufficiently extreme, the valuation multiplication inside `sri_value` can overflow, causing a runtime trap and degrading Serai chain availability.

Mitigation

Avoid the panic by using `checked_mul` and handling the overflow appropriately by returning `u64::MAX` to signal the overflow:

```
let denom = u128::from(10u64.pow(balance.coin.decimals()));
let quote_for_balance = u128::from(balance.amount.0)
    .checked_mul(quote_per_unit)
    .map(|x| x / denom)
    .unwrap_or(u128::MAX);
Amount(u64::try_from(quote_for_balance).unwrap_or(u64::MAX))
```

8.17 S1-772: Retirement-signal favors become unremovable after signal expiry or revocation

Attack scenario	**Stale favor records become undeletable and accumulate in storage**
Classification	VLN-9: Storage Exhaustion
Component	signals pallet
Tracking	https://github.com/serai-dex/serai/issues/772
Attack impact	T-7: Perform DoS attacks to halt or slow down block production
Severity	Low
Status	Acknowledged

Background

The signals pallet lets validators express protocol-level intent, specifically to retire the Serai protocol in favor of a new one or to permanently halt an external network.

Issue description

`revoke_favor_internal` validates the target signal before removing the stored favor record.

For retirement signals, `validate_signal` requires that:

- the retirement signal is still registered
- the retirement signal has not expired
- no retirement has already been locked in

That validation happens before `Favors::<T>::take(...)` is reached. As a result, if a validator previously favored a retirement signal and that signal is later revoked or expires, the validator can no longer remove their stored favor through `revoke_favor`.

```
Self::validate_signal(&signal)?;

if Favors::<T>::take((signal, with_network), validator) <
Some(Self::current_serai_session())
{
  // Stale favor is not considered to exist
  Err::<(), _>(Error::<T>::RevokingNonExistentFavor)?;
}
```

Risk

Honest users cannot remove their own favor records once the referenced retirement signal is expired or revoked. Stale `Favors` entries remain in storage indefinitely, bloating the storage.

Mitigation

Permit favor deletion even if the referenced retirement signal is expired or absent, or add automatic cleanup on retirement-signal revocation/expiry.

8.18 S0-731: serai-primitives - Missing validation of External Coin and Network association

Attack scenario	**An attacker wants to exploit a mismatch between coin and network**
Classification	VLN-15: Insecure Business Logic
Component	primitives crate
Tracking	https://github.com/serai-dex/serai/issues/731
Attack impact	T-9: Exploit a mismatch between coin and network
Severity	Info
Status	Mitigated

Background

The Serai primitives implementation define the correlation between networks and coins.

Issue description

The primitives crate defines `ExternalNetworkId` and `ExternalCoin` as logically associated types, but it does not enforce their consistency when constructing or processing external transfer instructions. For example, ExternalNetworkId::Ethereum is implicitly expected to be used with ExternalCoin::Ether, but no runtime check ensures that a coin actually belongs to the declared network. As a result, a transfer instruction may claim to send Ether on the Bitcoin network or XMR on Ethereum, violating domain constraints.

`src/coin.rs`

```
pub enum ExternalCoin {
    Btc,
    Ether,
    Dai,
    Xmr,
}

impl ExternalCoin {
    pub fn network(&self) -> ExternalNetworkId {
        match self {
            ExternalCoin::Bitcoin => ExternalNetworkId::Bitcoin,
            ExternalCoin::Ether | ExternalCoin::Dai =>
ExternalNetworkId::Ethereum,
            ExternalCoin::Monero => ExternalNetworkId::Monero,
        }
    }
}
```

`src/network_id.rs`:

```
pub enum ExternalNetworkId {
    Bitcoin,
    Ethereum,
    Monero,
}

impl ExternalNetworkId {
    pub fn coins(&self) -> &'static [ExternalCoin] {
        match self {
            ExternalNetworkId::Bitcoin => &[ExternalCoin::Btc],
            ExternalNetworkId::Ethereum => &[ExternalCoin::Ether,
ExternalCoin::Dai],
            ExternalNetworkId::Monero => &[ExternalCoin::Xmr],
```

```
        }
    }
}
```

However, beyond these mappings, the code does not enforce validity checks at runtime. In particular, there is no dedicated function like `validate_coin_for_network(coin, network)` that would reject mismatched pairs. The two mapping functions above simply provide the canonical pairing but do not guard other code paths against misuse.

We found no validity checks in the functionality that processes `ExternalNetworkId` and/or `ExternalCoin`.

This structural gap could allow semantically incorrect or misleading messages to propagate if not being caught at higher level APIs outside of serai-primitives.

Risk

An attacker could potentially craft a transfer instruction that combines mismatched network and coin identifiers. For example, they could declare a Bitcoin network withdrawal of Ether, potentially tricking off-chain relayers or bridge infrastructure into executing incorrect asset movements. This mismatch could lead to loss of funds, misrouting of value, or security violations in external systems that rely on this data. If no higher-level enforcement exists, these inconsistencies can silently propagate, violating assumptions across modules or between on-chain and off-chain logic.

Mitigation

Introduce runtime checks wherever `ExternalCoin` and `ExternalNetworkId` appear together to ensure the coin belongs to the declared network. Refactor the type system to encapsulate coin-network relationships more tightly, possibly by introducing a struct that pairs a network with its valid coins and disallows invalid constructions at the type level. Validate incoming instructions during decoding or execution to reject mismatched combinations.

8.19 S0-757: Genesis liquidity signature verification uses latest auxiliary keys instead of set-bound keys

Attack scenario	**A validator rotates aux keys after signing, invalidating `oraclize_values` until signatures are recollected.**
Classification	VLN-15: Insecure Business Logic
Component	serai-genesis-liquidity pallet
Tracking	https://github.com/serai-dex/serai/issues/757
Attack impact	T-7: Perform DoS attacks to halt or slow down block production.
Severity	Info
Status	Mitigated

Background

`serai-genesis-liquidity-pallet` finalizes the genesis bootstrap by accepting an unsigned `oraclize_values` call, initializing pools, and setting validator allocation parameters for subsequent security phases. The `validate_unsigned` logic for `oraclize_values` is responsible for authenticating that the submitted values were approved by the expected Serai validator participants and that the call is only accepted at the correct genesis timing window.

Issue description

The validity of the `oraclize_values` unsigned call is being checked inside the `validate_unsigned` function, as part of the `serai-genesis-liquidity-pallet`. The signer reconstruction uses `T::ValidatorSets::auxiliary_keys(validator, NetworkId::Serai)`, which returns the latest auxiliary key state rather than the key snapshot bound to the decided validator set. This makes signature verification depend on mutable post-decision key updates: if a validator rotates their Serai auxiliary key after signatures are collected, a previously valid aggregate signature can fail verification.

Risk

The primary risk is liveness/reliability degradation during genesis finalization. The off-chain orchestration may repeatedly produce signatures that fail on-chain until it re-collects signatures using the newly updated keys. This is not a direct authorization bypass under current assumptions, but it creates an avoidable dependency on mutable state in a sensitive initialization step and can delay successful oraclization.

Mitigation

Use set-bound Serai keys for signer reconstruction during `oraclize_values` validation instead of latest auxiliary keys. Concretely, derive participating public keys from the selected validator-set snapshot (or an equivalent immutable genesis signer snapshot) so verification is stable with respect to post-decision key rotations.

8.20 S0-774: Permissionless sweep of residual balances via zero-amount removal

Attack scenario	**An attacker calls remove_genesis_liquidity with zero genesis liquidity and sweeps any assets sitting at the genesis address.**
Classification	VLN-15: Insecure Business Logic
Component	genesis-liquidity pallet
Tracking	https://github.com/serai-dex/serai/issues/774
Attack impact	T-4: Exploit a vulnerability in Serai resulting in stolen liquidity
Severity	Info
Status	Risk accepted

Background

The `genesis-liquidity` pallet manages Serai's genesis liquidity program: it tracks the initial liquidity commitments, later accepts validator-oraclized reference values, and governs how those positions can be unwound after the genesis phase.

Issue description

Any account can call `remove_genesis_liquidity` with `genesis_liquidity.amount = 0` post-oraclization, even if it has never held genesis liquidity tokens. The `burn_fn` in the coins pallet succeeds for zero amounts without checking the caller's balance:

```
fn decrease_balance_internal(from: SeraiAddress, balance: Balance) -> Result<(),
Error<T, I>> {
  let coin = &balance.coin;

  let new_amount =
    (Self::balance(from, coin) - balance.amount).ok_or(Error::<T,
I>::NotEnoughCoins)?;

  if new_amount == Amount(0) {
    Balances::<T, I>::remove(from, coin);
  } else {
    Balances::<T, I>::set(from, coin, new_amount);
  }
  Ok(())
}
```

After the no-op DEX removal, the function reads the entire balance at the genesis address, and transfers everything to the caller. External coins are fully swept, SRI is split between the caller and a permanent burn based on the trickle feed position.

Risk

Under normal operations the genesis addresses are zeroed after each removal, so exploitability depends on coins arriving through external transfers or future protocol changes. However, the permissionless nature (no genesis token ownership required) means a front-runner could monitor for accidental or programmatic deposits and claim them instantly. The attack has zero cost beyond transaction fees.

Mitigation

Add an early return or error in the post-oraclization path when `genesis_liquidity.amount == 0`:

```
if genesis_liquidity.amount == Amount(0) {
    return Ok(());
}
```

Alternatively, require that the caller's genesis token balance is non-zero before proceeding past the burn.

8.21 S0-776: Unreviewed Security-Relevant TODOs

Attack scenario	An attacker does...
Classification	VLN-X: YZW
Component	Runtime configuration / Pallet X / etc
Tracking	https://github.com/serai-dex/serai/issues/776
Attack impact	T-X: YZW
Severity	Info
Status	Open

Background

The Serai node relies on `serai-env` as the single abstraction for reading secrets and configuration at startup. This crate is shared across the node and coordinator and sits on the critical path for key material ingestion.

Issue description

The codebase contains TODO comments that mark known security-relevant gaps as deferred work. These were identified during review but could not be fully assessed because the intended replacement behavior has not yet been implemented. The `serai_env::var` function carries two such TODOs: One acknowledging that secrets should be sourced from a proper secret store rather than the process environment, and one noting that variables should be unset after being read. Neither is implemented; the function currently delegates directly to `std::env::var`. The security properties of the final design depend entirely on decisions captured only in these comments.

Risk

The risk is scoped to the gap between the current placeholder implementation and whatever replaces it. Until that design is finalized and implemented, the security posture of secret handling in the node cannot be fully evaluated.

Mitigation

Resolve the TODOs in `common/env/src/lib.rs` before production deployment. Track outstanding security-relevant TODOs as first-class issues so they are not deferred indefinitely.

Appendix A: Vulnerability categories

Category	Description
VLN-1: Insufficient Existential Deposit	Inadequate existential deposits can lead to denial-of-service attacks by filling the blockchain storage as accounts below the deposit are reaped to conserve space
VLN-2: XCM Exploitation	Denial-of-service attacks via XCM can disrupt parachains or the relay chain, necessitating proper handling of untrusted incoming XCM messages and correct implementation of `XCMFeeManager`
VLN-3: Reliance on On-Chain Randomness	Weak on-chain randomness can be exploited to predict or control outcomes of critical functionalities, as seen with the insecure randomness collective flip pallet
VLN-4: Incorrect Benchmarking	Incorrect or missing benchmarking can cause overweight blocks and spam attacks by underestimating computational complexity or database access, leading to exceeded block execution times
VLN-5: Unsafe Arithmetic	Unsafe arithmetic can cause overflows and underflows, leading to unexpected states, as demonstrated by the overflow vulnerability in the `receive_messages_proof` extrinsic
VLN-6: Unsafe Conversion	Unsafe conversion from larger to smaller-sized values can result in precision loss and unexpected states, exemplified by the potential overflow in u128 to u64 conversions in Polkadot SDK
VLN -7: Reachable Panic	Reachable panics, caused by functions like `panic` or `unwrap`, and decoding without depth limits, can lead to critical severity issues, especially in `on_initialize` or `on_finalize` hooks
VLN-8: Insecure Cryptography	Use of insecure cryptographic libraries or primitives can compromise a Polkadot-SDK-based chain at various development stages, requiring extensive reviews for changes to cryptographic elements
VLN-9: Storage Exhaustion	Adversaries can attempt to fill blockchain storage cheaply, making node operation unsustainable. Charging deposits for on-chain storage helps mitigate this issue
VLN-10: Abusable unsigned and `Pays::No` calls	Unsigned extrinsics or those returning `Pays::No` can be exploited to spam the blockchain, as seen in the broker pallet issue
VLN-11: Outdated Crates	Outdated Rust crates, containing invalid or buggy code, pose security risks to the ecosystem and must be monitored and updated regularly
VLN-12: Consumers/Providers/Sufficients	Complexity in entity existence logic often leads to mishandled reference counts, causing vulnerabilities like preventing the creation of precomputed asset-conversion pools
VLN-13: Incorrect Slashing Logic	Ineffective or partial slashing fails to deter malicious behavior, undermining incentives against misbehavior in critical roles

VLN-14: Replay Issues	Replay issues can enable spamming or double-spending attacks when nonces are mishandled, such as in crowdloan contributions with `ExistenceRequirement::AllowDeath`
VLN-15: Insecure Business Logic	Business logic vulnerabilities stem from protocol flaws enabling valid transaction exploitation. Key issues include improper transaction validation, incentive misalignment, and unhandled edge cases.
VLN-16: Incorrect Rewarding Logic	Ineffective or partial rewarding fails to incentivize honest behavior, undermining decentralization

Appendix B: Code maturity categories

Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Appendix C: Technical services

Security Research Labs delivers extensive technical expertise to meet your security needs. Our comprehensive services include software and hardware evaluation, penetration testing, red team testing, incident response, and reverse engineering. We aim to equip your organization with the security knowledge essential for achieving your objectives.

SOFTWARE EVALUATION We provide assessments of application, system, and mobile code, drawing on our employees' decades of experience in developing and securing a wide variety of applications. Our work includes design and architecture reviews, data flow and threat modelling, and code analysis with targeted fuzzing to find exploitable issues.

BLOCKCHAIN SECURITY ASSESSMENTS We offer specialized security assessments for blockchain technologies, focusing on the unique challenges posed by decentralized systems. Our services include smart contract audits, consensus mechanism evaluations, and vulnerability assessments specific to blockchain infrastructure. Leveraging our deep understanding of blockchain technology, we ensure your decentralized applications and networks are secure and robust.

POLKADOT ECOSYSTEM SECURITY We provide comprehensive security services tailored to the Polkadot ecosystem, including parachains, relay chains, and cross-chain communication protocols. Our expertise covers runtime misconfiguration detection, benchmarking validation, cryptographic implementation reviews, and XCM exploitation prevention. Our goal is to help you maintain a secure and resilient Polkadot environment, safeguarding your network against potential threats.

TELCO SECURITY We deliver specialized security assessments for telecommunications networks, addressing the unique challenges of securing large-scale and critical communication infrastructures. Our services encompass vulnerability assessments, secure network architecture reviews, and protocol analysis. With a deep understanding of telco environments, we ensure robust protection against cyberthreats, helping maintain the integrity and availability of your telecommunications services.

DEVICE TESTING Our comprehensive device testing services cover a wide range of hardware, from IoT devices and embedded systems to consumer electronics and industrial controls. We perform rigorous security evaluations, including firmware analysis, penetration testing, and hardware-level assessments, to identify vulnerabilities and ensure your devices meet the highest security standards. Our goal is to safeguard your hardware against potential attacks and operational failures.

CODE AUDITING We provide in-depth code auditing services to identify and mitigate security vulnerabilities within your software. Our approach includes thorough manual reviews, automated static analysis, and targeted fuzzing to uncover critical issues such as logic flaws, insecure coding practices, and exploitable vulnerabilities. By leveraging our expertise in secure software development, we help you enhance the security and reliability of your codebase, ensuring robust protection against potential threats.

PENETRATION & RED TEAM TESTING We perform high-end penetration tests that mimic the work of sophisticated adversaries. We follow a formal penetration testing methodology that emphasizes repeatable, actionable results that give your team a sense of the overall security posture of your organization.

SOURCE CODE-ASSISTED SECURITY EVALUATIONS We conduct security evaluations and penetration tests based on our code-assisted methodology that lets us find deeper vulnerabilities, logic flaws, and fuzzing targets than a black-box test would reveal. This gives your team a stronger assurance that the significant security-impacting flaws have been found and corrected.

SECURITY DEVELOPMENT LIFECYCLE CONSULTING We guide organizations through the Security Development Lifecycle to integrate security at every phase of software development. Our services include secure coding training, threat modelling, security design reviews, and automated security testing implementation. By embedding security practices into your development processes, we help you proactively identify and mitigate vulnerabilities, ensuring robust and secure software delivery from inception to deployment.

REVERSE ENGINEERING We assist clients with reverse engineering efforts that are not associated with malware or incident response. We also provide expertise in investigations and litigation by acting as experts in cases of suspected intellectual property theft.

HARDWARE EVALUATION We evaluate new hardware devices ranging from novel microprocessor designs, embedded systems, mobile devices, and consumer-facing end products to core networking equipment that powers Internet backbones.

VULNERABILITY PRIORITIZATION We streamline vulnerability information processing by consolidating data from compliance checks, audit findings, penetration tests, and red team insights. Our prioritization and automation strategies ensure that the most critical vulnerabilities are addressed promptly, enhancing your organization's security posture. By systematically categorizing and prioritizing risks, we help you focus on the most impactful threats, ensuring efficient and effective remediation efforts.

SECURITY MATURITY REVIEW We conduct comprehensive security maturity reviews to evaluate your organization's current security practices and identify areas for improvement. Our assessments cover a wide range of criteria, including policy development, risk management, incident response, and security awareness. By benchmarking against industry standards and best practices, we provide actionable insights and recommendations to enhance your overall security posture and guide your organization toward achieving higher levels of security maturity.

SECURITY TEAM INCUBATION We provide comprehensive support for building security teams for new, large-scale IT ventures. From Day 1, our ramp-up program offers essential security advisory and assurance, helping you establish a robust security foundation. With our proven track record in securing billion-dollar investments and launching secure telco networks globally, we ensure your new enterprise is protected against cyberthreats from the start.

HACKING INCIDENT SUPPORT We offer immediate and comprehensive support in the event of a hacking incident, providing expert analysis, containment, and remediation. Our services include detailed forensics, malware analysis, and root cause determination, along with actionable recommendations to prevent future incidents. With our rapid response and deep expertise, we help you mitigate damage, recover swiftly, and strengthen your defenses against potential threats.