Security Research Labs

Moonbeam security audit for runtime 3900, 4000 and 4100

Threat model and hacking assessment report

v1.0, 3rd of February 2026

Prepared for:
Moonbase One SEZC

Content

Disclaimer **2**
Timeline **3**
Integrity Notice **4**
1 Executive summary **5**
1.1 Engagement overview 5
1.2 Observations and risk 5
1.3 Recommendations 5
2 Evolution suggestions **6**
3 Motivation and scope **7**
4 Methodology **8**
4.1 Threat modeling and attacks 8
4.2 Security design coverage check. 10
4.3 Implementation check 10
4.4 Remediation support 11
5 Findings summary **12**
5.1 Risk profile 12
5.2 Issue summary 13
6 Detailed findings **14**
6.1 S2-61: Insecure migration code might unlock staked funds in case of migration failure 14
6.2 S1-64: Collective precompile: gas calculation error in members and is_member 15
6.3 S1-63: Collective precompile: Missing check for proposal length in close 16
6.4 S1-60: Lack of congestion control mechanisms allow for queue bloating 17
7 Bibliography **19**
Appendix A: Vulnerability categories **21**
Appendix B: Code maturity categories **22**
Appendix C: Technical services **23**

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 were discovered in the codebase exhaustively and that following all suggestions may not ensure future code to be bug free.

Version:	v1.0
Client:	Moonbeam
Date:	3rd of February 2026
Audit Team:	Cayo Fletcher-Smith Constantin Schwarz Marc Heuse

Timeline

The Moonbeam source code has undergone one initial baseline audit alongside a subsequent continuous security assurance, started in May 2021 by Security Research Labs. Continuous security checks have been in place. As shown in Table 1, specific runtime audits for the Moonbeam runtimes 3900, 4000 and 4100 have been performed from July to December 2025.

Date	Event
1st July 2025	Start of Moonbeam audit for runtimes 3900, 4000 and 4100
31st December 2025	End of Moonbeam audit for runtimes 3900, 4000 and 4100
3rd February 2026	Delivered: Security audit report for Moonbeam runtimes 3900, 4000 and 4100

Table 1: Audit timeline

Integrity Notice

1 Executive summary

1.1 Engagement overview

This report documents the results of a continuous security assurance audit of Moonbeam that Security Research Labs performed for the runtime versions 3900, 4000 and 4100.

During this study, Moonbeam provided access to relevant documentation and effectively supported the research team. We verified the protocol design, concept documentation, and relevant available source code of Moonbeam.

Security Research Labs has conducted continual comprehensive security assurance for Moonbase One SEZC since our baseline audit in June 2021 and has been providing specialized audit services for Polkadot and Polkadot SDK projects since 2019. This audit focused on assessing the changes introduced into Moonbeam's codebase at runtime versions 3900, 4000 and 4100 for resilience against hacking and abuse scenarios. Key areas of scrutiny included: fee calculation mechanisms; Ethereum virtual machine recreation accuracy in Substrate; precompiled contracts; governance mechanisms and curves; runtime configurations; and pallet integrations. We prioritized reviewing critical functionalities and conducting thorough security tests to ensure the robustness of Moonbeam's platform. We collaborated closely with Moonbeam, utilizing full access to source code and documentation to perform a rigorous assessment.

1.2 Observations and risk

The research team identified one medium severity issue and three low severity issues. Moonbeam has acknowledged and, in cooperation with Security Research Labs, remediated the majority of identified issues in a timely manner. This shows that Moonbeam is developed with a strong sense of security, and that developers are capable of reacting to security issues quickly. One issue, namely S1-63, relies on resolution in cooperation from Parity.

1.3 Recommendations

Manual code audits provide an in-depth insight into the security posture of a project and are essential for maintaining a high security standard in the long term. However, the effectiveness of these audits can be improved significantly by performing automated tests, such as a continuous fuzzing campaign or by integrating static analysis tooling into the CI/CD pipeline.

2 Evolution suggestions

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

Perform dynamic analysis. The Moonbeam team received an up-to-date harness and instructions on how to run our fuzzing campaign on their own servers and customize it to their needs, however so far this has not been started. Additionally, the existing harness would benefit from more invariant tests being added. We recommend allocating resources towards integrating fuzzing into the overall testing process.

Use static analysis. Using static analysis tools to detect security flaws in the codebase is essential for improving code security. These tools, such as Dylint and Semgrep for the Rust ecosystem, analyze the code without executing it, identifying vulnerabilities, coding errors, and compliance issues early in the development process. This proactive approach helps developers address potential security issues before they reach production, ensuring a more secure and reliable codebase.

Create an incident response plan and validate it regularly. 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. Regular testing of the incident response plan through simulated breach drills and tabletop exercises ensures that the team is familiar with the plan and capable of executing it under stress. 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.

Prepare for emergency governance actions. As responding to an incident in the Polkadot ecosystem usually involves time sensitive governance proposals on the relay chain, it is crucial that they succeed in a timely manner. Incident response plans must account for scenarios in which the practical deadline for such a proposal is not the end of the voting period, but rather a much shorter period after which a malicious action performed by the attacker is executed. To ensure the success of incident response proposals, key community members of the relay chain should be involved in the incident response plans described above.

Assess builders and ecosystem risk before upgrades. Before enacting breaking changes, dependent projects and community builders should be assessed for potential adverse side-effects. This would include regularly reviewing the project implementations built on Moonbeam and understanding the impact certain changes may have on those codebases. Examples of breaking changes where this is present include adjustment in opcode fee adjustments, account heuristic adjustments, and certain precompile breaking changes.

3 Motivation and scope

This report presents the results of the security audit for Moonbeam runtime 3900, 4000 and 4100 from July to December 2025. It is important to note that the closed findings from previous engagements are not included in this document.

Moonbeam is a blockchain network, built using the Polkadot SDK, deployed on the Polkadot relay-chain, designed to be Ethereum compatible while extending its feature set with governance, staking and cross-chain integration support. As a result, hacking scenarios for Moonbeam include attacks from both the Polkadot ecosystem and relevant Ethereum attacks.

Moonbeam has cultivated a decentralized ecosystem centric around providing Ethereum application support within the broader Substrate community. This vision has been achieved by:

1. Providing support for Ethereum-style RPC-calls which allow existing Ethereum applications to be compatible with Substrate via Moonbeam.
2. Mapping existing Substrate accounts to the 20-byte Ethereum address format which allows users and smart contract applications to interact with accounts uniformly in both Ethereum and Moonbeam.
3. Integrating runtime gas metering to emulate the transaction fee mechanisms present in the Ethereum blockchain, while remaining compliant with the Substrate weight system. This allows Solidity smart contracts to exist on Moonbeam without requiring prior Substrate benchmarking.
4. Implementing an extensive precompile feature set, mapping core Substrate pallets to Solidity interfaces accessible via Ethereum style calls.

Like other Polkadot parachains, Moonbeam is built using the Polkadot SDK written in Rust, a memory safe programming language. Polkadot SDK based chains utilize three technologies: a WebAssembly (WASM) based runtime, decentralized communication via libp2p, and a block production engine.

Moonbeam's runtime consists of multiple modules compiled into a WASM Binary Large Object (blob) that is stored on-chain. Nodes execute the runtime code either natively or will execute the on-chain WASM blob.

In the initial baseline assurance audit, Security Research Labs collaborated with the Moonbeam development team to create an overview containing modules in scope and their audit priority. Following our baseline assurance, we gradually expanded our scope as new features became available and collaboratively outlined audit priority with the Moonbeam development team.

4 Methodology

We applied the following four-step methodology when performing feature and PR reviews for the Moonbeam network: (1) threat modeling, (2) security design coverage checks, (3) implementation baseline check, and finally (4) remediation support.

4.1 Threat modeling and attacks

The goal of the threat model framework is to determine specific areas of risk in Moonbeam network. 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. This section introduces how risk is defined and provides an overview of the identified threat scenarios.

The risk level is categorized into low, medium, and high and considers both the hacking value and the damage that could result from successful exploitation. The risk of a threat scenario is calculated by the following formula:

$$Risk = Damage \times Hacking\ Value = \frac{Damage \times Incentive}{Easiness}$$

The *Hacking Value* is similarly categorized into low, medium, and high and considers the incentive of an attacker, as well as the effort required by an adversary to successfully execute the attack. The hacking value is calculated as follows:

$$Hacking\ Value = \frac{Incentive}{Easiness}$$

While an incentive describes what an adversary might gain from performing an attack successfully, easiness estimates the complexity of this same attack. The degrees of incentive and easiness are defined as follows:

Incentive:

- Low: Attacks offer the hacker little to no gain from executing the threat
- Medium: Attacks offer the hacker considerable gains from executing the threat
- High: Attacks offer the hacker high gains by executing this threat

Easiness:

- High: Attacks are easy to execute. They require neither elaborate technical knowledge nor considerable amounts of resources
- Medium: Attacks are difficult to execute. They might require bypassing countermeasures, the use of expensive resources, or a considerable amount of technical knowledge
- Low: Attacks are difficult to execute. The attacks might require in-depth technical knowledge, vast amounts of expensive resources, bypassing countermeasures, or any combination of these factors

Incentive and Easiness are divided according to Table 2.

Hacking Value/Likelihood	Low Incentive	Medium Incentive	High Incentive
Low Easiness	Low	Medium	Medium
Medium Easiness	Medium	Medium	High
High Easiness	Medium	High	High

Table 2: Hacking value measurement scale

Hacking scenarios are classified by the risk they pose to the system. Conversely, the *Damage* describes the negative impact that a given attack, if performed successfully, would have on the victim. The degrees of damage are defined as follows:

Damage:

- Low: Risk scenarios would cause negligible damage to the Moonbeam network
- Medium: Risk scenarios pose a considerable threat to Moonbeam's functionality as a network
- High: Risk scenarios pose an existential threat to Moonbeam network functionality

Damage and Hacking Value are divided according to Table 3.

Risk	Low hacking value	Medium hacking value	High hacking value
Low damage	Low	Medium	Medium
Medium damage	Medium	Medium	High
High damage	Medium	High	High

Table 3: Risk measurement scale

After applying the framework to the Moonbeam system, different threat scenarios according to the CIA triad were identified.

The CIA triad describes three security promises that can be violated by a hacking attack, namely confidentiality, integrity, and availability.

Confidentiality:

Confidentiality threat scenarios concern sensitive information regarding the blockchain network and its users. Native tokens are units of value that exist on the blockchain. Confidentiality threat scenarios include, for example, attackers abusing information leaks to steal native tokens from nodes participating in the Moonbeam ecosystem and claiming the assets (represented in the token) for themselves.

Integrity:

Integrity threat scenarios aim to disrupt the functionality of the entire network by undermining or bypassing the rules that ensure that Moonbeam transactions/operations are fair and equal for each participant. Undermining Moonbeam's integrity often comes with a high monetary incentive. For example, an attacker can double-spend or mint tokens for themselves. Other threat scenarios do not yield an immediate monetary reward but could rather damage Moonbeam's functionality and, in turn, its reputation. For example, unexpected or undocumented discrepancies between the virtual machine implementations in Moonbeam and the Ethereum network might be perceived as bugs by the users operating on the Moonbeam network.

Availability:

Availability threat scenarios refer to compromising the availability of data stored by the Moonbeam network as well as the availability of the network itself to process normal transactions. Important threat scenarios regarding the availability for blockchain systems include denial-of-service (DoS) attacks on the domain operators, stalling the transaction queue, and spamming.

Table 4 provides a high-level overview of the hacking risks concerning Moonbeam with the identified example threat scenarios and attacks, as well as their respective hacking value and effort. We guide our code review by internal threat modelling and confirming invariants with developers (our initial threat model is available online at [1]). Given the ongoing nature of this engagement, each component review warranted its own focused internal threat model which, when necessary, was thoroughly

discussed with the development team. The complete list of threat scenarios identified along with attacks that enable them are described in the threat model deliverable. By thinking in terms of threat scenarios and attacks during code review or feature ideation, many issues can be caught or even avoided altogether.

The threats were classified using the CIA security triad model, mapping threats to the areas: (1) Confidentiality, (2) Integrity, and (3) Availability.

Security promise	Hacking value	Example threat scenarios	Hacking effort	Example attack ideas
Confidentiality	Medium	Compromise a user's private key	High	Targeted attacks to compromise a user's private key
				Social engineering
Integrity	High	Censor certain transactions	Medium	Block transactions by overweight extrinsics
		Bypass fees		Exploit bug to waive fees
		Tamper collator nomination		Stall collator nomination
		Fabricate false transaction		Replay transactions
Availability	High	Stall block production	Low	Spam overweight extrinsics
		Clutter chain storage		Abuse cheap storage mechanisms
		Spam the network with bogus transactions		Clutter XCM queue via XCMP spam

Table 4: Risk overview

4.2 Security design coverage check.

Next, we reviewed the Moonbeam 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?

4.3 Implementation check

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

To effectively review the Moonbeam codebase and new features, we derived our code review strategy based on two aspects: First, the key areas of interest and priority detailed by the Moonbeam development team. Second, our own internal threat models created for each feature review. For each identified threat, hypothetical attacks were developed and mapped to their corresponding threat category, as outlined in Chapter 4.1.

Prioritizing potential risk for the network, the code was assessed for present protection 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 to assess the security of the Moonbeam codebase.

While static and dynamic testing establishes a baseline assurance, the focus of this audit was on manual code review of the Moonbeam codebase. The approach of feature reviews was to trace the intended functionality of modules in scope and to assess whether an attacker can bypass/misuse/abuse these components or trigger unexpected behavior on the blockchain. Since the Moonbeam codebase is entirely open source, it is realistic that an adversary could analyze the source code while preparing an attack.

4.4 Remediation support

The final step is supporting Moonbeam 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 a private GitHub repository [2]. We also used a private Slack channel for asynchronous communication and status updates. In addition, biweekly jour fixe meetings were held to provide detailed updates and address open questions.

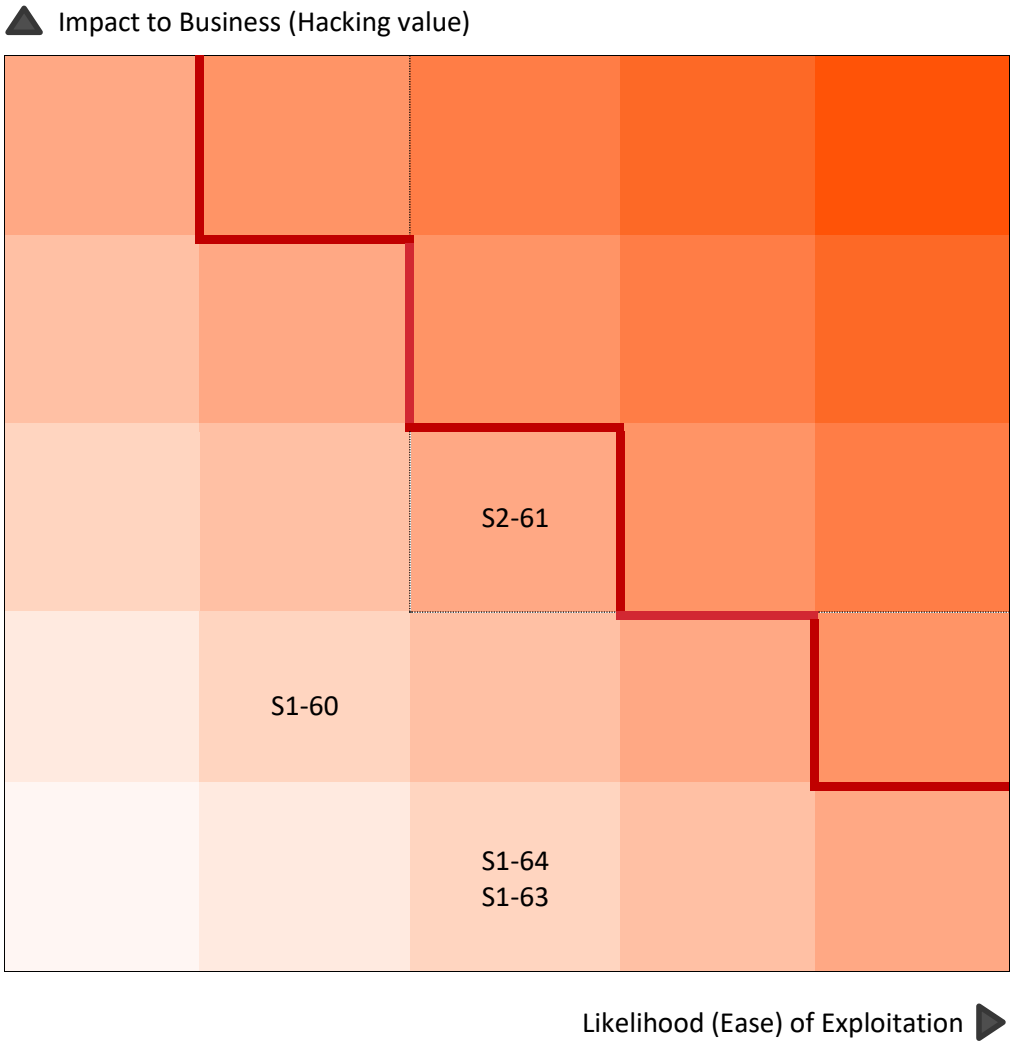
5 Findings summary

We identified one medium severity and three low severity issues during our analysis of the runtime modules in scope in the Moonbeam codebase. The finding is listed in Table 5.

High	0
Medium	1
Low	3
Total Issues	**4**

5.1 Risk profile

The chart below summarizes vulnerabilities according to business impact and likelihood of exploitation, increasing to the top right. The red margin separates the high-critical issues from medium/low/informational ones.



5.2 Issue summary

ID: Issue	Severity	Status
S2-61: Insecure migration code might unlock staked funds in case of migration failure	Medium	Mitigated [3]
S1-64: Collective precompile: gas calculation error in members and is_member	Low	Mitigated [4]
S1-63: Collective precompile: Missing check for proposal length in close	Low	Open
S1-60: Lack of congestion control mechanisms allow for queue bloating	Low	Mitigated [5]

Table 5: Findings overview

6 Detailed findings

6.1 S2-61: Insecure migration code might unlock staked funds in case of migration failure

Attack scenario	An attacker transfer funds and avoids slashing by forcing the account to have a higher number of freezes than there are variants of RuntimeFreezeReason.
Classification	VLN-15: Insecure Business Logic
Component	moonbeam/pallets/parachain-staking/src/lib.rs
Tracking	https://github.com/moonbeam-foundation/sr-moonbeam/issues/61
Attack impact	Staked funds could become transferable, potentially allowing users to bypass intended slashing or staking constraints.
Severity	Medium
Status	Mitigated [3]

Background

The Currency trait, responsible for dealing with underlying assets, has been deprecated [6]. This migrations logic aims to migrate all locked funds of collators and delegators into the newer, supported, Fungible trait. For the scope of this issue, both Currency and Fungible are functionally the same. The migration logic aims to safely unlock locked delegator funds and set them to frozen while the migration is on-going.

Issue description

The migration code has a fatal flaw: the funds stay unlocked in case the freezing fails. Because errors are handled early, the overall transaction will still succeed.

The `check_and_migrate_lock` function first removes existing locks before it freezes the same amount again using the fungible trait [7].

In case the `set_freeze` function errors, the original lock is not reapplied, and an error is returned. This error is either ignored [8] or only written to the debug log. In both cases the overall transaction will still succeed, allowing the staked funds to remain unlocked and unfrozen.

```
if amount > BalanceOf::<T>::zero() {
   // Remove any existing lock
   T::Currency::remove_lock(lock_id, account);
   // Set the freeze
   T::Currency::set_freeze(&freeze_reason.into(), account, amount)?;
}
```

Risk

Staked funds could be unlocked and transferred, potentially avoiding slashes. The `set_freeze` function would error if an account had a higher number of freezes than there are variants of `RuntimeFreezeReason`. We did not find a way to reach this state during the audit, it might be possible, however, for an account to be in this situation due to past or future code changes, or inaccurate manual storage writes.

Mitigation

The freeze should be applied before the lock is removed, so in case of a migration failure, the funds remain locked.

6.2 S1-64: Collective precompile: gas calculation error in members and is_member

Attack scenario	An attacker could call members or **`is_member`** after a configuration change where **`MaxMembers`** differs from **`MaxProposals`**, causing gas to be calculated incorrectly.
Classification	VLN-15: Insecure Business Logic
Component	moonbeam/precompiles/collective/src/lib.rs
Tracking	https://github.com/moonbeam-foundation/sr-moonbeam/issues/64
Attack impact	Gas may be under- or overcharged, leading to inefficient resource usage or minor economic inconsistencies.
Severity	Low
Status	Mitigated [4]

Background

The Collective Precompile allows developers to interact with Substrate's collective pallet directly through a Solidity interface, enabling on-chain governance actions from smart contracts. It supports scoped collectives. The `is_member` function is used to determine a user's permission to participate in actions such as voting, proposing and executing [9].

Issue description

Gas accounting in the members and `is_member` functions relies on the wrong configuration constant, which could lead to inaccurate gas charges if future runtime changes diverge proposal and member limits. In the fn members and `fn is_member` functions [10], the `MaxProposals` value is used instead of `MaxMembers` to record gas costs for database access.

```
handle.record_db_read::<Runtime>(
 20 * (<Runtime as pallet_collective::Config<Instance>>::MaxProposals::get()
 as usize),
)?;
```

Risk

Right now, the runtime config (stagenet) is the same number for both `MaxProposals` and `MaxMembers`, so there is little immediate risk – although when configuration changes in the collective pallet this will result in under or overcharging gas for these operations.

Mitigation

Use the `MaxMembers` value to calculate gas in `fn members` and `fn is_members`.

6.3 S1-63: Collective precompile: Missing check for proposal length in close

Attack scenario	An attacker could supply a length_bound smaller than the actual proposal length when calling close, causing the extrinsic to execute with incorrect parameters.
Classification	VLN-15: Insecure Business Logic
Component	moonbeam/precompiles/collective/src/lib.rs
Tracking	https://github.com/moonbeam-foundation/sr-moonbeam/issues/63
Attack impact	Gas could be undercharged and the proposal closure could fail, leading to failed operations and inefficient resource usage.
Severity	Low
Status	Open

Background

The Collective Precompile allows developers to interact with Substrate's collective pallet directly through a Solidity interface, enabling on-chain governance actions from smart contracts. It supports scoped collectives. The `close` function should safely close an existing proposal after enough votes have been reached [9].

Issue description

The `close` function does not validate that the provided length bound matches the actual proposal size. It takes `length_bound` as a parameter, which should be greater than or equal to the length of the SCALE-encoded proposal. However, there's no check in the precompile to ensure `length_bound` is indeed greater than or equal to the actual proposal length [11].

```
  pub fn close(
    origin: OriginFor<T>,
    proposal_hash: T::Hash,
    #[pallet::compact] index: ProposalIndex,
    proposal_weight_bound: Weight,
    #[pallet::compact] length_bound: u32,
  ) -> DispatchResultWithPostInfo {
      ensure_signed(origin)?;

      Self::do_close(proposal_hash, index, proposal_weight_bound, length_bound)
  }
```

Risk

A malicious caller could provide a `length_bound` that's smaller than the actual proposal length, which will cause the following issues during the call execution on the Substrate side:

1. The close extrinsic weight will be undercharged, as its weight function relies on this parameter
2. The close extrinsic will error out with `WrongProposalLength`

Mitigation

Perform the checks on the correctness of `length_bound` or use a safe overestimation parameter in the precompile logic that is passed to the pallet instead to avoid undercharging gas.

The mitigation proposal has been acknowledged and is being tracked in the polkadot-sdk repository.

6.4 S1-60: Lack of congestion control mechanisms allow for queue bloating

Attack scenario	An attacker could continuously submit messages to the bridge, exploiting the lack of congestion management and insufficient fee scaling, to bloat the **`OutboundQueue`**.
Classification	VLN-9: Storage Exhaustion
Component	moonbeam /runtime/moonbase/src/bridge_config.rs
Tracking	https://github.com/moonbeam-foundation/sr-moonbeam/issues/60
Attack impact	Legitimate messages could be delayed or blocked, causing extreme delivery delays and potential disruption of cross-chain operations.
Severity	Low
Status	Mitigated [5]

Background

The `xcm-bridge` adds support for opening and closing bridges between chains across different global consensuses. It manages bridge metadata and message lanes, enabling chains to coordinate cross-consensus messaging through a dedicated, stable `LaneId`. The `xcm-bridge` relies on the `LocalXcmChannelManager` implementation to suspend the message bridging in case congestion has been detected. This is crucial in maintaining the availability of bridging components.

Issue description

Congestion is checked and suspension performed in the `on_bridge_message_enqueued()` function after a message has been enqueued. Moonbeam and Moonriver rely on the default implementation for `LocalXcmChannelManager`, that does not have any logic implemented to suspend or resume the bridge, and simply assumes it is always in an uncongested state.

In the runtime configuration for `LocalXcmChannelManager` it is configured as () meaning no management actions are performed upon congestion [12]. This prevents the bridge from automatically limiting outgoing messages via `is_congested` if the number of unconfirmed messages grows too large [13]. As seen in the following code block, `is_congested` always returns false:

```
fn is_congested(_with: &Location) -> bool {
    false
}
```

Furthermore, bridging fees do not get prohibitively more expensive linear to the queue size of `OutboundMessage`. It is currently not clear if the basic dynamic fees applied in Moonbeam are enough to adequately account for storage bloating of `OutboundMessages`. However, it must be assumed that they are not.

Dynamic fees in-theory make it infeasible to continuously bloat the `OutboundMessages` queue, since submitting vast quantities of messages via `xcm-transact` will increase the block usage. Although it may be possible that dangerous levels of queue bloating may still be achieved while the block capacity remains at < 50%, as it is generally possible that a small message with a low fee on the sending side still results in an expensive execution on the receiving chain. That would result in a disconnect between fee-adjustment and queue size.

Risk

The lack of congestion management means that under extreme stress new messages will still be queued, regardless of the capabilities of relayers to deliver messages and execute them on the destination chain. The lack of prohibitive congestion-based fees may allow attackers to bloat the

`OutboundQueue` such that legitimate messages are unable to be delivered and experience extreme delays.

Mitigation

We recommend at least enforcing the baseline congestion thresholds already present in the bridge pallets. Additionally, mechanisms should be implemented to gradually increase the cost of bridging based on the current levels of congestion in the `OutboundQueue`.

7 Bibliography

[1] [Online]. Available: https://securityresearchlabs.sharepoint.com/:x:/s/Purestake/EQBP8RMQ9IdCn5DqBsQC_boB5IboSsW5snvEX15VfY70ow.

[2] [Online]. Available: https://github.com/moonbeam-foundation/sr-moonbeam.

[3] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/pull/3306.

[4] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/pull/3540.

[5] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/commit/e98e23ccd999ab680d14731669adb39a54b2a9d5.

[6] [Online]. Available: https://paritytech.github.io/polkadot-sdk/master/frame_support/traits/tokens/currency/index.html.

[7] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/blob/fdaca0bdf5cfc38353779349626a8f101a87c790/pallets/parachain-staking/src/lib.rs#L1638.

[8] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/blob/fdaca0bdf5cfc38353779349626a8f101a87c790/pallets/parachain-staking/src/lib.rs#L1711.

[9] [Online]. Available: https://docs.moonbeam.network/builders/ethereum/precompiles/features/governance/collective/.

[10] [Online]. Available: https://github.com/datahaven-xyz/datahaven/blob/db99f62d3ba90263fe3c7fb148b7defaec9bda8b/operator/precompiles/collective/src/lib.rs#L322.

[11] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/blob/3c26cddc38721c16e2bd50d6a197536742133f8d/precompiles/collective/src/lib.rs#L250.

[12] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/blob/f9665416460695617685c750f18b96982996fc6b/runtime/moonbase/src/bridge_config.rs#L191.

[13] [Online]. Available: https://github.com/moonbeam-foundation/polkadot-sdk/blob/eb18eec247946d6c7210135f35515d1d01f57737/bridges/modules/xcm-bridge-hub/src/exporter.rs#L215-L218.

[14] [Online]. Available: https://github.com/srlabs/substrate-runtime-fuzzer.

[15] [Online]. Available: https://github.com/moonbeam-foundation/sr-moonbeam/issues/60.

[16] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/blob/ca05e285ce6c16146da1a7a8f1f1db9b33aab687/pallets/moonbeam-foreign-assets/src/lib.rs#L334.

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.

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 moelling, 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.