Security Research Labs

Moonbeam security audit for runtime 2900, 3000 and 3100

Threat model and hacking assessment report

4th of September 2024

Prepared for:
Moonbase One SEZC

Content

Disclaimer

This report describes the findings and core conclusions derived from the audit carried out by Security Research Labs within the agreed-on timeframe and scope (described 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 evolution suggestions described in Chapter 2 may not ensure all future code to be bug free.

Version:	Final, V1.1	
Prepared For:	Moonbase One SEZC	
Date:	4 September 2024	
Prepared By:	Marc Heuse	marc@srlabs.de
	Bruno Produit	bruno@srlabs.de
	Cayo Fletcher-Smith	cayo@srlabs.de
	Daniel Schmidt	schmidt@srlabs.de

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 , specific runtime audits for Moonbeam runtimes 2900, 3000 and 3100 have been performed from January to July 2024.

Date	Event
January 1, 2024	Start of Moonbeam audit for runtimes 2900, 3000 and 3100
September 4, 2024	Delivered: Security audit report for Moonbeam runtime 2900, 3000 and 3100

Table 1: Security assurance timeline

1 Executive Summary

1.1 Engagement Overview

This work describes the result of the security audit for Moonbeam runtime 2900, 3000 and 3100 from January to July 2024. Security Research Labs is a consulting firm that has been providing specialized audit services in the Polkadot ecosystem since 2019, including for the Substrate and Polkadot projects.

During this assessment, the Moonbeam team provided access to relevant documentation and supported the research team effectively. The code of Moonbeam was verified to assure that the business logic of the product is resilient to hacking and abuse.

Security Research Labs has conducted continual comprehensive security assurance for Moonbase One SEZC, in partnership with the organization, since our baseline audit in June 2021. Our security assurance for this audit for runtimes 2900, 3000 and 3100 focused on assessing Moonbeam's codebase 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.

Our 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 Moonbeam's platform. Throughout the review process, Security Research Labs collaborated closely with Moonbeam, utilizing full access to source code, documentation and the development team to perform a rigorous assessment.

1.2 Observations and Risk

The research team identified two medium level severity issues, which concerned mostly fee calculations and weights. Moonbeam has acknowledged and, in cooperation with Security Research Labs, remediated the identified issues.

1.3 Recommendations

The deployment and roll-out of runtime upgrades pose risks due to potential human error in deployment methods, rather than the upgrade logic itself. We recommend revisiting and strengthening the framework of checks and best practices for runtime migrations, coupled with high-level threat analysis and support from Security Research Labs, to reduce any associated migration risks.

2 Evolution suggestions

2.1 Risk reduction frameworks for enacting runtime migrations

The deployment and roll-out of runtime upgrades opens the Moonbeam network up to edge-cases associated with human error. Considering such edge-cases do not necessarily exist in the upgrade logic, but instead the methods of deployment, vulnerabilities are difficult to foresee and mitigate through the current development testing strategy and security review process.

We recommend revisiting the framework of checks and best practices implemented when performing migrations. We may offer high-level threat analysis and support to ensure these processes are as robust as possible.

2.2 Secure development improvement suggestions

We recommend to further strengthen the security of the Moonbeam blockchain by implementing the following recommendations:

Perform threat modeling. Performing threat modeling for all new features and major updates before coding is crucial for better code security. This practice allows developers to identify potential security threats and vulnerabilities early in the design phase, enabling them to implement appropriate mitigations from the outset. Including the threat model in the pull request description ensures that the entire team is aware of the identified risks and the measures taken to address them, promoting a proactive security culture and enhancing the overall robustness of the codebase. Additionally, it helps the audit team to identify gaps in the threat model and focus their assessment. The threat model should be part of the pull request 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.

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.

2.3 Security alignment and workflow

Throughout the continued collaboration between Security Research Labs and the Moonbeam development team there existed, although infrequent, notable instances of inadequate alignment. Such instances include inaccuracies regarding the perceived status of open issues and the acceptance of associated mitigations.

We recommend the prioritization of security ambassador programs, to ensure alignment and effective dissemination of progress alongside enhancing proactive communication and collaboration on security initiatives.

3 Motivation and scope

This report presents the result of the security audit for Moonbeam runtime 2900, 3000 and 3100 from January to July 2024. 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 Substrate, 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 allows 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 Substrate-framework written in Rust, a memory safe programming language. Substrate-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

This report details the results of our security audit on the Moonbeam runtimes 2900, 3000 and 3100 between January to July 2024 with the aim of creating transparency in the following steps: threat modeling, security design coverage checks, reviewing runtime changes and finally remediation support.

4.1 Threat modeling and attacks

The goal of the threat model framework is to be able 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 the security testing. This section introduces how risk is defined and provides an overview of the identified threat scenarios. The *Hacking Value*, categorized into *low*, *medium*, and *high*, considers the incentive of an attacker, as well as the effort required by an attacker to successfully execute the attack. The hacking value is calculated as:

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

While *incentive* describes what an attacker might gain from performing an attack successfully, *effort* estimates the complexity of this same attack. The degrees of incentive and effort 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.

Effort:

- Low: 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.
- High: 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 effort are divided according to Table 2.

Hacking value	Low incentive	Medium incentive	High incentive
High effort	Low	Medium	Medium
Medium effort	Medium	Medium	High
Low effort	Medium	High	High

Table 2: Hacking value measurement matrix

Hacking scenarios are classified by the risk they pose to the system. The risk level, also categorized into low, medium, and high, considers the hacking value, as well as the damage that could result from successful exploitation. The risk of a threat scenario is calculated by the following formula:

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

Damage describes the negative impact that a given attack, 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's 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 management matrix

After applying the framework to the Moonbeams 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, availability.

Confidentiality:

Confidentiality threat scenarios concern sensitive information regarding the blockchain network and its users. 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 for themselves.

Integrity:

Integrity threat scenarios threaten 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 high monetary incentives, for example, if an attacker can double spend or arbitrarily mint tokens. Other threat scenarios may not yield an immediate monetary reward, but instead threaten to damage Moonbeam's functionality and reputation. For example, unexpected or undocumented discrepancies between the virtual machine implementations in Moonbeam and the Ethereum network.

Availability:

Availability threat scenarios refer to compromising both the accessibility of data stored by the network and the ability for the network to process transactions. Important threat scenarios regarding availability for blockchain systems include denial of service (DoS) attacks on collator nodes, stalling the transaction queue, and halting block production.

Table 4 offers an overview of the hacking risks associated with Moonbeam, detailing example threat scenarios, potential attacks examples, and their corresponding hacking value and effort. An initial threat model, shared with the team on 22nd of June 2021 (available online at [1]), served as the foundation for all subsequent security reviews. 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.

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

4.2 Security design coverage check.

Moonbeam feature designs were reviewed for coverage against relevant hacking scenarios. For each scenario, the following two aspects were investigated:

a. **Coverage**. Is each potential security vulnerability sufficiently covered?

b. **Underlying assumptions**. Which assumptions must hold true for the design to effectively reach the desired security goal?

4.3 Implementation check

Moonbeam features were tested 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 both: the key areas of interest and priority detailed by the Moonbeam development team; alongside 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 by risk, 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, the auditors:

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. Manual code audits, fuzz testing, and manual tests were performed where appropriate.

3. Ensured the code did not contain any vulnerabilities that could be used to execute the respective attacks, otherwise, 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 testing and dynamic tests (e.g., fuzz testing) to assess the security of the Moonbeam codebase.

While static testing, fuzz testing and dynamic tests establish baseline assurance, the focus of our security assurance is primarily manual code review. 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 a malicious actor would analyze the source code while preparing an attack.

Fuzz testing – is a technique to identify issues in code that handles untrusted input, which in Moonbeam's case are extrinsics and more specifically precompile calls. Fuzz testing works by taking some valid input for a method under test, applying a semi-random mutation, and then invoking the method under test again with this semi-valid input.

When applying an input we monitor for adverse side-effects, such as crashes, that indicate broken logic. We also include invariants that, if broken, do not necessarily indicate broken logic but instead imply failed developer assumptions and business logic.

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.

4.4 Remediation support

The final step is supporting Moonbeam with the remediation process of identified issues. Each finding was documented and disclosed to the Moonbeam development team with accompanying mitigation recommendations.

Our remediation recommendations are specifically tailored with an understanding of Moonbeam's business strategy and core success factors. As such we often support multiple remediation strategies to accommodate Moonbeam's unique requirements and risk management plan.

Once the remediation is live, the fix is verified by our auditors to ensure that it mitigates the issue and does not introduce alternative bugs.

Throughout our collaboration, findings were disclosed via their private GitHub repository. We also engaged in asynchronous communication and status updates – in addition, bi-weekly jour fixe meetings were held to provide detailed updates and address open questions.

5 Dynamic analysis assessment

Throughout our security review from January to July 2024, we maintained an in-depth coverage guided fuzzing campaign to expand our assurance to include a variety of Moonbeam specific edge-cases. By integrating fuzz testing into our overall review strategy, we uncovered issues that would likely not have been detected through manual code analysis.

Harness creation and genesis configuration – We chose our in-house developed and opensource tool Ziggy as our fuzzing orchestration tool. We designed a heavily customized harness to ensure that many of the Ethereum-style features were accessible to the fuzzer in a meaningful way.

Collaboration – Alongside our own maintenance of the Moonbeam fuzzing harness and in the interest of transparency and close-collaboration; we shared our in-house fuzzing tools and techniques with the Moonbeam team. This helped us identify areas of improvement (e.g., coverage priority) alongside enabling greater test coverage for the Moonbeam development team.

We place great emphasis on iteratively improving the efficiency of fuzz testing in our Moonbeam review workflow. This encompasses the following key areas:

1. Improvement of genesis configuration to ensure our fuzzer has adequate capabilities to interface with the entirety of the Moonbeam codebase.
2. Coverage analysis to identify modules our fuzzer struggles to reach, which is often due to additional genesis requirements.
3. Injection of new seeds into the corpus, often derived from test cases and known call-sequences.
4. Continued invariant design derived from known developer assumptions (e.g., from smoke-tests) and key areas of priority.
5. Crash analysis and issue triage to appropriately assess the impact, risk and validity of automated findings.

Continually maintaining our fuzzing process has yielded a multitude of security findings and proves critical in our ability to identify edge-cases specific to the Moonbeam ecosystem.

5.1 Pallet coverage

To improve pallet coverage, we manually derive call-sequences, known as seeds, from Moonbeam test cases. These seeds are introduced to the corpus to ensure the fuzzer understands how to reach greater depths in the associated functionality.

Table 5 below details the line coverage achieve by our fuzzing campaign on each individual pallet.

Component	Code path	Coverage
asset-manager	pallets/asset-manager/src	22.94%
erc20-xcm-bridge	pallets/erc20-xcm-bridge/src	94.24%
ethereum-xcm	pallets/ethereum-xcm/src	66.67%
moonbeam-lazy-migrations	pallets/moonbeam-lazy-migrations/src	54.63%
moonbeam-oribters	pallets/moonbeam-oribters/src	28.26%
parachain-staking	pallets/parachain-staking/src	32.29%
proxy-genesis-companion	pallets/proxy-genesis-companion/src	69.23%

xcm-transactor	pallets/xcm-transactor/src	48.24%

Table 5: Moonbeam pallets fuzzing coverage

5.2 Precompile coverage

To allow coverage into the precompiled contracts we implement a genesis configuration to map the fuzzer's `AccountId` to an Ethereum-compatible 20-byte address. Coverage is then improved by deriving seeds from precompile test-cases. Existing limitations in precompile coverage are found in modules that rely on relay-chain feedback, for example, `relay-data-verifier`.

Table 6 details the line coverage achieved by our fuzzing campaign on each individual pallet.

Component	Code path	Coverage
asset-erc20	precompiles /asset-erc20/src	4.86%
author-mapping	precompiles/author-mapping/src	93.24%
balances-erc20	precompiles/balances-erc20/src	78.35%
batch	precompiles/batch/src	86.99%
call-permit	precompiles/call-permit/src	74.29%
collective	precompiles/collective/src	88.99%
conviction-voting	precompiles/conviction-voting/src	46.93%
crowdloan-rewards	precompiles/crowdloan-rewards/src	82.98%
gmp	precompiles/gmp/src	7.11%
identity	precompiles/identity/src	20.84%
parachain-staking	precompiles/parachain-staking/src	66.55%
preimage	precompiles/preimage/src	93.55%
proxy	precompiles/proxy/src	50.65%
randomness	precompiles/randomness/src	39.59%
referenda	precompiles/referenda/src	42.6%
relay-data-verifier	precompiles/relay-data-verifier/src	1.82%
relay-encoder	precompiles/relay-encoder/src	52.8%
utils-evm	precompiles/utils/src/evm	91.67%
utils-solidity	precompiles/utils/src/solidity	84%
xcm-transactor	precompiles/xcm-transactor/src	52.23%
xcm-utils	precompiles/xcm-utils/src	75.82%

xtokens	precompiles/xtokens/src	60%

Table 6: Moonbeam precompiles fuzzing coverage

6 Findings summary

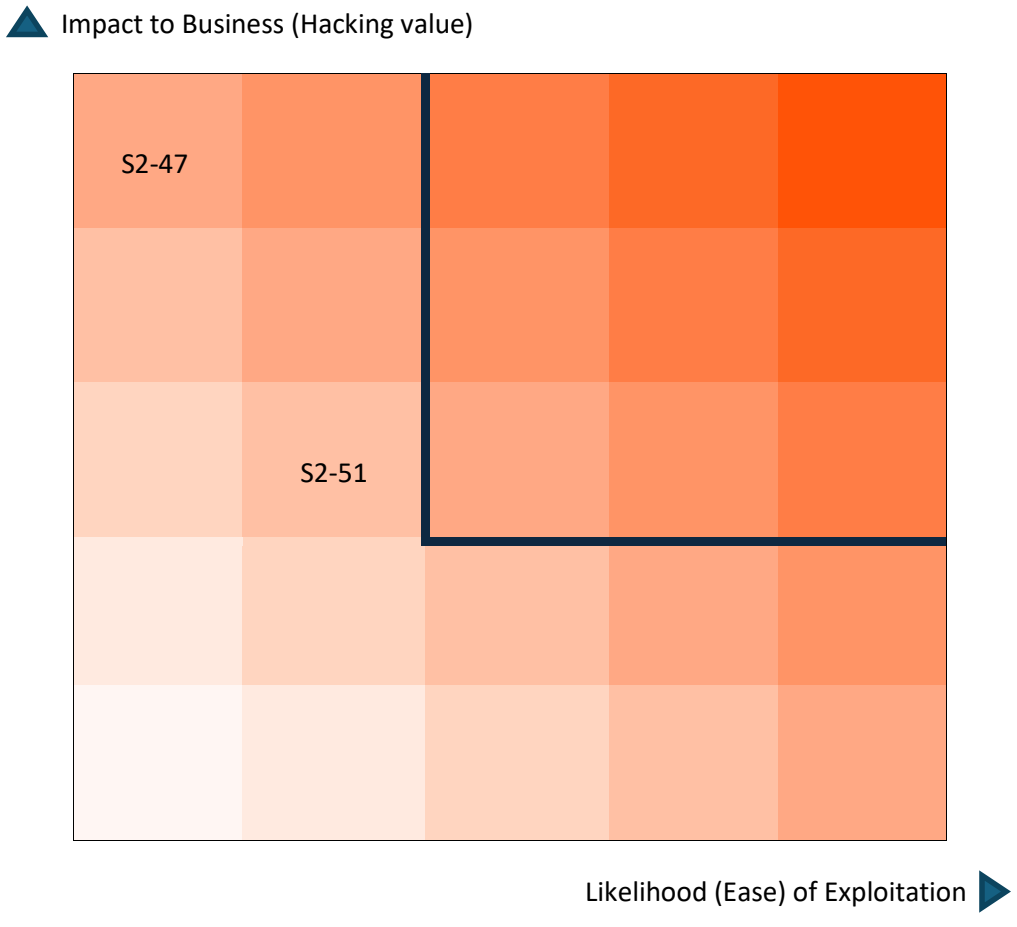
For our security audit from January to July 2024, Security Research Labs identified 2 medium security issues. These findings, in Table 7., are the culmination of various security testing processes implemented during our commitment to enhancing and preserving Moonbeam's security.

Critical	0
High	0
Medium	2
Low	0
Total Issues	**2**

Please note that in our methodology, critical severity issues refer to high severity issues that could be exploited immediately by an attacker on already deployed infrastructure.

6.1 Risk profile

The chart below summarizes vulnerabilities according to business impact and likelihood of exploitation, increasing to the top right.



6.2 Issue summary

Tracking	Issue	Severity	Status
S2-47 [2]	Incorrect gas weighting may stall block production	Medium	Mitigated [3] [4]
S2-51 [5]	Incorrect configuration of runtime weights	Medium	Mitigated [6]

Table 7: Code review issue summary

7 Detailed findings

7.1 S2-47: Incorrect gas weighting may stall block production

Attack scenario	An attacker brute-forces colliding storage keys to unbalance the Trie then executes two computational storage reads via `extcodesize` over XCM
Location	runtime/moonbeam/src/lib.rs
Attack impact	Block production could be entirely halted without remediation since XCM implies forced execution
Severity	Medium
Status	Mitigated [3] [4]
Tracking	[2]

Background and Context
Moonbeam uses Frontier to generate gas costs for EVM code execution then converts the gas calculation to an equivalent substrate weight by using `WeightPerGas` for `ref_time` and `GasLimitPovSizeRatio` [7] for `proof_size`. Since opcodes, specifically `extcodesize`, rely on this static conversion from gas to weigh, computationally expensive storage reads may result in inaccuracies in the recorded `proof_size`.

Problem Details
When performing the `extcodesize` opcode, which returns the size of bytecode stored at a `H160` address, `account_code_metadata()` attempts to perform a storage read to `AccountCodesMetadata` [8] which contains the structure `CodeMetadata` and subsequently the code `size: u64` [9] for a given storage key (`H160` address).

If the data is non-existent in `AccountCodesMetadata`, another storage read [10] is performed to `AccountCodes` which houses the full bytecode as `code`, this is subsequently checked by and returned as `code.len` by `from_code()` [11] to satisfy the `extcodesize` call. If `code` exists within `AccountCodes`, but not inside `AccountCodesMetadata`, the metadata for the account (e.g `CodeMetadata.size:u64`) is updated immediately [12].

When deploying a new smart contract, the `create_account()` function ensures that `AccountCodesMetadata` is correctly initialized [13], thereby making it impossible to abuse the gas to weight conversion by deploying large contract bytecode without metadata then forcing large storage reads to the `code` in `AccountCodes`.

An attacker can circumvent the inability to perform reads to `AccountCodes` by calling `extcodesize` on a non-existent address, this will return nothing on the lookup for `size:u64` in `CodeMetadata`, which subsequently forces a storage read to `AccountCodes`.

To abuse this an attacker may brute force a number of Sybil `H160` addresses such that after the SCALE-encoding and `Blake2_128Concat` hashing occurs the resulting storage keys collide for the first 32-bits.

This scenario focuses on unbalancing the Trie nodes through partial storage key collision. This may cause storage reads to require significantly more computation when traversing child nodes. If accomplished an attacker may call `extcodesize` with a specific, non-existent, address which would perform two computationally expensive storage reads to `AccountCodesMetadata` and `AccountCodes`.

Risk
To consume the full block-size `proof_size` limit (5-megabytes) an attacker must first ensure that each `extcodesize` lookup results in a storage read totalling in a `proof_size` of 2800-bytes.

An attacker may reach the `proof_size` limit, provided they can force the `extcodesize` storage reads to traverse an additional 5.5 full Trie-nodes, since each full node, in base-16, requires $32 \times 16 = 516$ bytes alongside some small overhead.

An attacker may purposefully abuse this by first bloating the storage Trie then performing the two-storage reads caused by `extcodesize` on a non-existent address over XCM.

If this is exploited over XCM: the nature of forced execution will ensure overweight blocks are unavoidable, resulting in denial-of-service to Moonbeam.

Security Research Labs acknowledged that this issue is conducive to the underlying substrate storage schema, and as such could apply to many different storages reads. Considering this, we worked closely to provide recommendations that heavily reduce the risk of this specific instance to an acceptable baseline.

Recommendation
We initially recommended deriving the gas value from the maximum execution time (`ref_time`) and maximum PoV size usage (`proof_size`) so that the worst-case scenario (including trie unbalancing) is accounted for.

Following close collaboration with the Moonbeam development team we agreed on the implementation of halting mechanisms to pause the processing of incoming XCM messages, alongside the introduction of an execution-error if `CodeMetadata` is absent, instead of relying on the subsequent storage read to `AccountCodes`. This remediation strategy effectively mitigates the particular risks associated with the double storage-read and exploitation of this theoretical issue via XCM.

Moonbeam has since integrated the XCM halting mechanism [3] and removed the secondary `AccountCodes` storage read from the execution path [4]. This mitigation will be live in runtime 3200.

7.2 S2-51: Incorrect configuration of runtime weights

Attack scenario	Malicious or inadvertent calls to incorrectly weighted extrinsics may cause service issues
Location	runtime/*
Attack impact	Underweighted extrinsic calls may result block rejection upon relay-chain validation
Severity	Medium
Status	Mitigated [6]
Tracking	[5]

Background and Context
There are multiple instances of runtime weights being incorrectly configured to the default `SubstrateWeight<Runtime>` provided from Parity.

Problem Details
The runtime weights for pallets: `cumulus-pallet-parachain-system`, `cumulus-pallet-dmp-queue` and `pallet-message-queue` have been misconfigured in the `moonbase/`, `moonbeam/` and `moonriver/` runtimes.

An example of such misconfiguration is illustrated below:

```
impl cumulus_pallet_parachain_system::Config for Runtime {
	...
	type WeightInfo =
		cumulus_pallet_parachain_system::weights::SubstrateWeight<Runtime>;
}
```

Risk
Since the weights are not dependent on benchmarking performed in the context of the specific runtime. This may lead to overweight extrinsics causing blocks to be rejected by the relay-chain, resulting in availability issues.

Recommendation
We recommended performing all benchmarking in the context of the specific runtime and avoiding the use of default Substrate weights by including them in the `define_benchmarks!` block [14]. An example can be viewed in the Kusama runtime template [15].

Moonbeam remediated this issue by introducing accurate weighting for the pallets [6]. This will be live in runtime-3200.

Bibliography

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

[2] [Online]. Available: https://github.com/moonbeam-foundation/sr-moonbeam/issues/47.

[3] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/pull/2745.

[4] [Online]. Available: https://github.com/moonbeam-foundation/frontier/pull/224/files.

[5] [Online]. Available: https://github.com/moonbeam-foundation/sr-moonbeam/issues/51.

[6] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/pull/2909.

[7] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/blob/ef76ba7b98dab867c17579be68e8bca5bdf6e688/runtime/moonbeam/src/lib.rs#L411.

[8] [Online]. Available: https://github.com/moonbeam-foundation/frontier/blob/5d0e5c4c3e2bbeeb60bd0520c2ad6a92532a75da/frame/evm/src/lib.rs#L866.

[9] [Online]. Available: https://github.com/moonbeam-foundation/frontier/blob/5d0e5c4c3e2bbeeb60bd0520c2ad6a92532a75da/frame/evm/src/lib.rs#L610.

[10] [Online]. Available: https://github.com/moonbeam-foundation/frontier/blob/5d0e5c4c3e2bbeeb60bd0520c2ad6a92532a75da/frame/evm/src/lib.rs#L870.

[11] [Online]. Available: https://github.com/moonbeam-foundation/frontier/blob/5d0e5c4c3e2bbeeb60bd0520c2ad6a92532a75da/frame/evm/src/lib.rs#L615.

[12] [Online]. Available: https://github.com/moonbeam-foundation/frontier/blob/5d0e5c4c3e2bbeeb60bd0520c2ad6a92532a75da/frame/evm/src/lib.rs#L886.

[13] [Online]. Available: https://github.com/moonbeam-foundation/frontier/blob/5d0e5c4c3e2bbeeb60bd0520c2ad6a92532a75da/frame/evm/src/lib.rs#L857.

[14] [Online]. Available: https://docs.substrate.io/test/benchmark/#adding-benchmarks.

[15] [Online]. Available: https://github.com/paritytech/polkadot/blob/01fd49a7fafa01f133e2dec538a2ef7c697a26aa/runtime/kusama/src/lib.rs#L1578-L1587.