Security Research Labs

Kagome Security Audit

Threat model and hacking assessment report

V1.1, 2 May 2024

Marc Heuse marc@srlabs.de

Gabriel Arnautu gabriel@srlabs.de

Justin Angra justin@srlabs.de

Abstract. This work describes the result of the thorough and independent security assurance audit of the Kagome Polkadot host implementation of Quadrivium performed by Security Research Labs. 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 study, Quadrivium provided access to relevant documentation and supported the research team effectively. The code of Kagome in the v0.9.3 release was reviewed and verified to assure that the business logic of the product is resilient to hacking and abuse.

The research team identified several issues ranging from critical to low, many of which concerned the software development lifecycle, packaged build dependencies, business logic, memory management, and differential analysis compared to the Polkadot host implementation in Rust. Quadrivium, in cooperation with the auditors, already remediated most identified issues and is working on the remaining.

In addition to mitigating the remaining open issues, Security Research Labs recommends increasing the project's security maturity by establishing a documented process for implementing test cases and sufficient documentation to ensure parity with the Polkadot implementation in Rust. Additionally, continuous node tests with sanitizers enabled are highly recommended.

Content

1 Motivation and scope

Blockchains evolve in a trustless and decentralized environment, which by its own nature could lead to security issues. Ensuring availability and integrity is a priority for Kagome as it aims for a feature-identical implementation of the Polkadot host in C++20. As such, a security review of the project should not only highlight the security issues uncovered during the audit process, but also bring additional insights from an attacker's perspective, which the Quadrivium team can then integrate into their own threat modelling and development process to further enhance the security of the product.

Kagome is a blockchain node built to integrate seamlessly with the Substrate network. With a goal of providing further accessibility and utility outside the Rust ecosystem, Kagome code is written in C++20, a low-level, general-purpose, cross-platform programming language. Mainly, Substrate-based chains utilize three technologies: a WebAssembly (WASM) based runtime, decentralized communication via libp2p, and a block production engine.

The Kagome 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.

The core business logic of Kagome is to implement a node compatible with other Polkadot host implementations, in line with the Polkadot specification but as an alternative implementation so that the Polkadot ecosystem does not depend on a single client.

The assessment was performed on the v0.9.3 release of Kagome (https://github.com/qdrvm/kagome/releases/tag/v0.9.3).

Security Research Labs collaborated with the Quadrivium team to create an overview containing the scope and their audit priority. The in-scope components and their assigned priorities are reflected in Table 1. During the audit, Security Research Labs used a threat model to guide efforts on exploring potential security flaws and realistic attack scenarios.

Repository	Priority	Component(s)
https://github.com/qdrvm/kagome	High	Block Production
		GRANDPA
		BEEFY
		Trie and Storage
		Parachains Core
	Medium	Transaction Pool
		Synchronizer
		Host API
		Runtime API
		Networking
		Crypto
	Low	RPC
		Offchain Workers

	Telemetry

Table 1. In-scope Kagome components with audit priority

2 Methodology

This report details the baseline security assurance results for the Kagome Polkadot host implementation with the aim of creating transparency in four steps, treat modeling, security design coverage checks, implementation baseline check, and finally remediation support:

Threat Modeling. The threat model is considered in terms of *hacking incentives*, i.e., the motivations to achieve the goals of breaching the integrity, confidentiality, or availability of Kagome nodes. For each hacking incentive, hacking *scenarios* were postulated, by which these goals could be achieved. The threat model provides guidance for the design, implementation, and security testing of Kagome. Our threat modeling process is outlined in Chapter 3.

Security design coverage check. Next, the Kagome design was 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?

Implementation baseline check. As a third step, the current Kagome implementation was tested for openings whereby any of the defined hacking scenarios could be executed.

To effectively review the Kagome 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 3.

Prioritizing by risk, the codebase was assessed for present protections against the respective threats and attacks and the vulnerabilities that make these attacks possible. For each threat, the auditors:

1. Identified the relevant parts of the codebase, for example the relevant namespace and the deployment 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 discovered to the development team along with suggestions around mitigations.

Security Research Labs carried out a hybrid strategy utilizing a combination of code review and dynamic tests (e.g., fuzz testing) to assess the security of the Kagome codebase.

While fuzz testing and dynamic tests establish a baseline assurance, the focus of this audit was a manual code review of the Kagome codebase to identify logic bugs, design flaws, and best practice deviations. We reviewed version 0.9.3 of the Kagome repository up to the commit cbbee3b. The approach of the review was to trace the intended functionality of the in-scope modules 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 Kagome 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. 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., type confusion) in the method under test. In Kagome's case we focused our fuzzing effort on the network communication.

Remediation support. The final step is supporting Quadrivium 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 the auditors to ensure that it mitigates the issue and does not introduce other bugs.

During the audit, findings were shared via a private GitHub repository. Additionally, a private Telegram channel for asynchronous communication and status updates was used. Fortnightly meetings were held to provide detailed updates and address open questions.

3 Threat modeling and attacks

The goal of the threat model framework is to be able to determine specific areas of risk in Quadrivium's Kagome Polkadot host implementation. Familiarity with these risk areas can provide guidance for the implementation stack's design, 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 scale.

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 Kagome node and reputation.
- Medium: Risk scenarios pose a considerable threat to Kagome's functionality as a Polkadot host.
- High: Risk scenarios pose an existential threat to Kagome's Polkadot host functionality or reputation.

Damage and Hacking Value are divided according to Table 3.

Risk	Low hacking value	Medium hacking	High hacking
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 Kagome, 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. 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 Kagome ecosystem and claiming the assets (represented in the token) for themselves.

Integrity:

Integrity threat scenarios threaten to disrupt the functionality of the entire network by undermining or bypassing the rules that ensure that Kagome node transactions/operations are fair and equal for each participant. Undermining Kagome's integrity often comes with a high monetary incentive, like for example, if an attacker can double spend or mint tokens for themselves. Other threat scenarios do not yield an immediate monetary reward, but rather, could threaten to damage Kagome's functionality and, in turn, its reputation. For example, insufficient consistency with the Polkadot specification would violate the core promise of feature-for-feature compatibility with the other Polkadot host implementation.

Availability:

Availability threat scenarios refer to compromising the availability of data stored by the Kagome node, as well as the availability of the node itself to process normal transactions. Important threat scenarios regarding availability for blockchain systems include Denial of Service (DoS) attacks on participating nodes, stalling the transaction queue, and spamming. Table 4 provides a high-level overview of the hacking risks concerning Kagome with identified example threat scenarios and attacks, as well as their respective hacking value and effort. The complete list of threat scenarios identified along with attacks that enable them are described in the threat model deliverable *SRLabs-Kagome_Audit-Threat_Model-2023*. This list can serve as a starting point to the Quadrivium developers to guide their security outlook for future feature implementations. By thinking in terms of threat scenarios and attacks during code review or feature ideation, many issues can be caught or even avoided altogether.

For Kagome, the auditors attributed the most hacking value to the integrity class of threats. Since the efforts required to exploit this kind of issue are considered lower, we identified threat scenarios to the integrity of Kagome as of the highest risk

category. Undermining the integrity of the Kagome node means making unauthorized modifications to the system. Some of the scenarios can have a direct effect on the financials of the system. This can include market manipulation, gaining tokens for free, or as a vault stealing collateral without repercussions.

Security promise	Hacking value	Example threat scenarios	Hacking effort	Example attack ideas
Confidentiality	High	- Steal token from node	High	- Decrypt private information through bad usage of cryptographic protocol
Integrity	High	- Double spend tokens via getting the clients to accept a different chain - Undermine consensus mechanism to split chain - Tamper / manipulate blockchain history to invalidate transactions (e.g. a voting result)	Medium	- Double spend through a nothing-at-stake attack - Modify critical business data by exploiting logic error in the code - Transaction malleability attack - Consensus split
Availability	High	- Conduct (D)DoS attacks to disrupt network connectivity and operations - Undermine block production or consensus mechanism	Medium	- DoS validator nodes - Transaction spam - Find memory leaks

Table 4. Risk overview. The threats for Kagome's Polkadot host implementation were classified using the CIA security triad model, mapping threats to the areas: (1) Confidentiality, (2) Integrity, and (3) Availability.

4 Findings summary

We identified 20 issues - summarized in Table 5 - during our analysis of the in-scope node modules within the Kagome codebase that enable some of the attacks outlined above. In summary, 2 critical severity, 14 high severity, 1 medium severity and 3 low severity issues were found.

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, including a parachain or a non-incentivized testnet.

Issue	Severity	References	Status
Outdated libsecp256k1 in Hunter Package Manager	Critical	[1]	Mitigated [2]
Host Memory Exhaustion via Excessive Number of Invalid Deallocation Calls	Critical	[3]	Mitigated [4]
Race Condition Leading to Heap Use-After-Free in Kagome Node	High	[5]	Mitigated
Memory Allocation Crash Vulnerability in Kagome Node	High	[6]	Mitigated [7] [8]
Memory Read Overflow Crash in detachNode() Function	High	[9]	Mitigated [10]
ScaleDecoder: Unbound Allocation Leads to Timeout and Memory Exhaustion	High	[11]	Mitigated [12]
Numerous TODOs in Kagome Code - some untracked - Affecting Security and Stability	High	[13]	In Progress
Insufficient Codebase Comments	High	[14]	In Progress
Lack of Parachain Tests in Kagome Leads to Unverified Implementation Consistency with Polkadot	High	[15]	In Progress
Lack of BEEFY Tests in Kagome Leads to Unverified Implementation Consistency with Polkadot	High	[16]	In Progress
Lack of GRANDPA Tests in Kagome Leads to Unverified Implementation Consistency with Polkadot	High	[17]	In Progress
Missing Memory and CPU Time Limit for PVF Pre-Checking	High	[18]	Mitigated [19] [20]

Missing Deterministic Stack Depth Instrumentation of WASM Code	High	[21]	Mitigated [22]
Incorrect Implementation of Versioned Cryptographic Host Functions	High	[23]	Mitigated [24]
Behavioral differences in Memory allocator allows wasm code to distinguish between Kagome and Polkadot reference implementation, leading to consensus violation	High	[25]	Mitigated [4] [26]
Incorrect Behavior in Cryptographic Batch Verification	High	[27]	Mitigated [28]
Insecure TLS protocol Versions Enabled in SSL Configurations	Medium	[29]	Mitigated [30]
Absence of Secure-Validator Mode in Kagome Compared to Polkadot	Low	[31]	In Progress
Inadequate Memory Clearing Post-Private Key Usage	Low	[32]	Mitigated [33]
Buffer Read Overflow Risk in core/host_api/impl/crypto_extension.cpp	Low	[34]	Mitigated [35]

Table 5 Issue summary

5 Detailed findings

5.1 Outdated libsecp256k1 in Hunter package manager

Attack scenario	Outdated dependency can leak private key via timing attacks
Location	Cmake
Tracking	[1]
Attack impact	Compromise security of the cryptographic processes
Severity	Critical
Status	Mitigated [2]

As blockchain technology heavily relies on cryptographic processes, the usage of up to date and tested cryptographic libraries is critical.

Kagome is using an outdated version of the libsecp256k1 library from 2019. This version contains known vulnerabilities, specifically a “constant-time” issue could leak sensitive data.

Updating the library to its latest version, v0.4.1 December 2023, would mitigate this vulnerability. During the update process, other libraries should also be checked for their latest version, specifically OpenSSL and XXH3.

5.2 DOS via excessive number of invalid deallocation calls

Attack scenario	Cause memory exhaustion to crash Kagome nodes
Location	core/runtime/common
Tracking	[3]
Attack impact	Attacker could cause node denial-of-service
Severity	Critical
Status	Mitigated [4]

The WASM blob can allocate memory on the host node, which should be capped at 4GB per node, as per Polkadot’s implementation. The memory management of Kagome allows an unlimited number of deallocation calls, regardless of whether the memory region has been allocated before or not. This could allow a malicious WASM blob to exhaust the host-side memory, leading to a node crash.

The entire memory management module should be redesigned such as it follows the same implementation as Polkadot, including by-bug compatibility.

5.3 Race condition leading to use-after-free in Kagome node

Attack scenario	Kagome node crashes while doing network synchronization
Location	core/application/impl
Tracking	[5]
Attack impact	During the network synchronization process, the node might crash or have unexpected behavior
Severity	High

Status	Mitigated

When the node runs in synchronization mode, a race condition can be triggered which leads to a heap use-after-free. Such an issue may lead to undefined behavior within the affected thread.

Using smart pointers as “std::{unique,shared,weak}_ptr” would help mitigate this issue. Moreover, verifying if a pointer is NULL before its utilization would ensure safer usage.

5.4 Memory allocation crash vulnerability in Kagome node

Attack scenario	Crash the Kagome node by exploiting libp2p implementation
Location	libp2p
Tracking	[6]
Attack impact	Attacker can send a special crafted network packet to crash the Kagome node
Severity	High
Status	Mitigated [7] [8]

Libp2p is a dependency that handles the network communication in most blockchain systems. The C++ implementation of the libp2p developed by Quadrivium could allow an attacker to remotely cause a denial-of-service, crashing the Kagome node, by sending a specific network packet. The issue lies in incorrect memory allocation.

A fix in libp2p is required in the packet parsing module.

5.5 Read overflow crash in detachNode() function

Attack scenario	Kagome node crashes while doing network synchronization
Location	core/storage/trie/polkadot_trie
Tracking	[9]
Attack impact	During the network synchronization process, the node might crash or have unexpected behavior
Severity	High
Status	Mitigated [10]

When the node runs in synchronization mode, a crash can occur due to a memory read overflow. This is due to a comparison being made on two vectors with different sizes.

Because the comparison was meant to only check if a vector is a suffix of another vector, a better implementation that “std::equal” can be used.

5.6 Unbound allocation DOS in ScaleDecoder

Attack scenario	**DoS or missing block producing slot while decoding SCALE codec input**
Location	scale-codec-cpp
Tracking	[11]
Attack impact	An attacker could send a special crafted SCALE encoded transaction to make the node crash or spend too much time while decoding the transaction
Severity	High
Status	Mitigated [12]

SCALE is a light-weight data format used in the Substrate framework to encode the transactions. The Kagome implementation of SCALE codec is vulnerable to memory allocation issues for all the variable size structures (e.g.: std::vector, ResizableCollection, std::deque). This can make the node crash or take too much time (between 30 and 120 seconds) to decode the transaction, making the validators miss their validation slots.

To mitigate this vulnerability, the input validation should be improved by enforcing a maximum value for “input_count”.

5.7 Numerous TODOs in Kagome code – some untracked

Attack scenario	**Kagome node crashes during operation**
Location	Multiple modules
Tracking	[13]
Attack impact	During operation, the node may crash or skip important features due to a lack of implementation
Severity	High
Status	Open

The codebase has a high number of “TODO” comments (over 100), most of them not having an issue assigned in the project’s issue tracking system. Part of the TODOs might relate to missing important features or security enhancements. The lack of traceability and accountability for these tasks could impact Kagome both from a security and stability point of view.

The team should allocate the required amount of time to investigate the respective “TODO”s, add them in the tracking system, prioritize and assign them to their responsible teams. Moreover, policies should be put in place such as this issue will not arise again in the future.

5.8 Insufficient codebase comments

Attack scenario	**Kagome developers may introduce errors within complex modules during maintenance or development**
Location	Multiple modules
Tracking	[14]

Attack impact	Lack of comments for high complexity code may lead to changes that negatively affect stability and security
Severity	High
Status	Open

The Kagome project's codebase has a significant imbalance in the distribution of code comments, with many areas lacking adequate commentary. This discrepancy leads to uneven understanding and maintainability across different parts of the code. As an example, the BABE module has one comment for every 41 lines. In comparison, the Polkadot implementation of BABE has a ratio of 1:6.

The absence of comments in critical areas can result in increased difficulty for developers in understanding and modifying the code. This lack of clarity can lead to a higher potential for errors, inefficient coding practices, and challenges in future maintenance and updates. Furthermore, new contributors or team members may face a steep learning curve, potentially slowing down development progress.

Depending on code complexity, established recommendations for line-to-code rations vary between 1:10 and 1:25. When Quadrivium starts adding comments to the code, functions with high cyclomatic complexity or critical functionality should be prioritized. Also links to the polkadot-sdk reference implementation should be added to the comments.

5.9 Lack of parachain tests leads to potential implementation differences

Attack scenario	**Kagome cannot integrate with the Polkadot ecosystem due to an incompatible parachain implementation**
Location	Multiple modules
Tracking	[15]
Attack impact	An incompatible implementation may decrease trust in Kagome, lead to integration issues, and indicate security vulnerabilities
Severity	High
Status	Open

Kagome currently only implements Zombienet tests, omitting any parachain tests from Polkadot. This exclusion raises concerns regarding the verification of its implementation's consistency and compatibility with Polkadot's ecosystem. Parachain tests are crucial for ensuring that implementations like Kagome can integrate seamlessly with Polkadot's parachains, which are essential for the network's interoperability and scalability.

The absence of parachain tests in Kagome's testing framework poses a high risk of undetected discrepancies between Kagome and Polkadot's expected behavior, potentially leading to integration issues, security vulnerabilities, and decreased trust in Kagome's reliability as a Polkadot Runtime Environment implementation. This gap undermines the confidence in Kagome's ability to support the Polkadot ecosystem's multi-chain architecture effectively.

To address this issue, we recommend integrating existing parachain tests from Polkadot into Kagome's Testing Framework. Directly leveraging Polkadot's existing parachain tests ensures that Kagome is tested against the same criteria as the Polkadot network, promoting consistency and interoperability. This approach would likely be the most straightforward way to validate Kagome's compatibility with Polkadot.

5.10 Lack of BEEFY tests leads to potential implementation differences

Attack scenario	**Kagome cannot integrate with the Polkadot ecosystem due to an incompatible BEEFY implementation**
Location	core/consensus/beefy, core/network/beefy
Tracking	[16]
Attack impact	An incompatible implementation may decrease trust in Kagome, lead to integration issues, and indicate security vulnerabilities
Severity	High
Status	Partially mitigated/Mitigated/Open (+add reference)

Kagome omits implementing BEEFY tests from Polkadot. This exclusion raises concerns regarding the verification of its implementation's consistency and compatibility with Polkadot's ecosystem. BEEFY tests are crucial for ensuring that implementations like Kagome can integrate seamlessly with the consensus model of the Polkadot ecosystem.

The absence of BEEFY tests in Kagome's testing framework poses a high risk of undetected discrepancies between Kagome and Polkadot's expected behavior, potentially leading to integration issues, security vulnerabilities, and decreased trust in Kagome's reliability as a Polkadot Runtime Environment implementation. This gap undermines the confidence in Kagome's ability to support the Polkadot ecosystem's consensus architecture effectively.

To address this issue, we recommend integrating existing BEEFY tests from Polkadot into Kagome's Testing Framework. Directly leveraging Polkadot's existing BEEFY tests ensures that Kagome is tested against the same criteria as the Polkadot network, promoting consistency and interoperability. This approach would likely be the most straightforward way to validate Kagome's compatibility with Polkadot.

5.11 Lack of GRANDPA tests leads to potential implementation differences

Attack scenario	**Kagome cannot integrate with the Polkadot ecosystem due to an incompatible GRANDPA implementation**
Location	core/consensus/grandpa
Tracking	[17]
Attack impact	An incompatible implementation may decrease trust in Kagome, lead to integration issues, and indicate security vulnerabilities
Severity	High
Status	Partially mitigated/Mitigated/Open (+add reference)

Kagome partially implements GRANDPA tests from Polkadot. This exclusion of all test cases raises concerns regarding the verification of its implementation's consistency and compatibility with Polkadot's ecosystem. GRANDPA tests are crucial for ensuring that implementations like Kagome can integrate seamlessly with the consensus model of the Polkadot ecosystem.

The absence of sufficient GRANDPA tests in Kagome's testing framework poses a high risk of undetected discrepancies between Kagome and Polkadot's expected behavior, potentially leading to integration issues, security vulnerabilities, and decreased trust in Kagome's reliability as a Polkadot Runtime Environment implementation. This gap undermines the confidence in Kagome's ability to support the Polkadot ecosystem's consensus architecture effectively.

To address this issue, we recommend integrating all existing GRANDA tests from Polkadot into Kagome's Testing Framework. Directly leveraging Polkadot's existing GRANDPA tests ensures that Kagome is tested against the same criteria as the Polkadot network, promoting consistency and interoperability. This approach would likely be the most straightforward way to validate Kagome's compatibility with Polkadot.

5.12 Missing memory and CPU time limit for PVF pre-checking

Attack scenario	**Malicious PVF WASM blob can cause excessive CPU usage or memory exhaustion**
Location	core/parachain/pvf
Tracking	[18]
Attack impact	An attacker could submit a malicious PVF WASM blob via a parachain, leading to a crash of Kagome nodes
Severity	High
Status	Mitigated [19] [20]

When a parachain wants to upgrade to a new Proof of Validation Function, it must submit the PVF for pre-checking. The validators will run the PVF and submit an “include_pvf_check_statement”. Once the supermajority (> 2/3) of validators is confirming that the pre-checking could successfully be executed within certain limits in terms of CPU and memory usage, the upgrade can progress.

The current Kagome implementation does not impose an execution time and memory limit for the PVF pre-checking.

Attackers able to submit a malicious PVF WASM blob via a parachain can cause excessive CPU usage (and possibly preventing checking of other legitimate WASM blobs in a timely manner) and/or memory exhaustion (leading to a crash of Kagome nodes).

Our recommendation is to implement memory usage and CPU time limits and to move the pre-check logic into a separate process which can be terminated when it exceeds the defined limits.

5.13 Missing deterministic stack depth instrumentation of WASM code

Attack scenario	**Kagome validators can get slashed**
Location	core/parachain/pvf
Tracking	[21]
Attack impact	Inconsistencies in the stack limit between Kagome and Polkadot can open disputes, which could lead to the slashing of the Kagome nodes
Severity	High
Status	Mitigated [22]

The polkadot-sdk reference implementation has a deterministic stack depth limit to ensure that a PVF will run out of stack space at the same point in time/recursion depth, regardless of variations in the wasmtime version or the used architecture. This has not yet been implemented in Kagome and this discrepancy between the two implementations may lead to disputes between honest validators running Kagome and others running polkadot-sdk.

It is possible that this may get exploitable even without a malicious PVF if one legitimate parachain has an unbounded recursion vulnerability. In that case a malicious collator can create a block which exceeds the stack depth limit enforced in polkadot-sdk but is still considered valid by Kagome. If this block is getting backed (which is possible as soon as two Kagome validators happen to be in the same backing group), it will be submitted to the chain and later disputed by all validators running polkadot-sdk.

The consistency of the stack limit is critical for consensus safety since all honest validators must be able to agree on whether a given PoV is valid or not and any discrepancies in that regard may lead to disputes and slashing. Since a majority of the validator set will likely continue running the reference implementation of polkadot-sdk in the foreseeable future, validators running Kagome will likely end up on the losing side of disputes caused by this.

To mitigate this vulnerability, a reimplementation of the stack depth instrumentation is needed. The implementation must be 100% compatible with the reference implementation to ensure that the limit is reached at exactly the same point in time/recursion depth.

5.14 Incorrect implementation of versioned cryptographic host functions

Attack scenario	**Disputes between honest host validators can be created**
Location	core/host_api/impl
Tracking	[23]
Attack impact	An attacker can submit a PVF which can distinguish between Kagome and Polkadot nodes, opening disputes after the execution fails on one of the host implementations.
Severity	High
Status	Mitigated [24]

The polkadot-sdk reference implementation contains multiple versions of certain cryptographic primitives with slightly different behavior. For example, sr25519, ECDSA, secp256k1 have an implementation for two different versions in Polkadot.

The Kagome implementation does not contain this versioning, but rather the second version is a wrapper for the first one.

A malicious PVF can distinguish between the Kagome and the Polkadot-sdk host implementation, effectively triggering disputes (and slashing) of honest validators running different host implementations.

The Kagome team must reimplement the host environment (including deprecated old versions of host functions) in a by-bug comparable way.

5.15 Differences in memory allocator leading to consensus violation

Attack scenario	**Disputes between honest host validators can be created**
Location	core/runtime/common
Tracking	[25]
Attack impact	An attacker can submit a PVF which can distinguish between Kagome and Polkadot nodes, opening disputes after the execution fails on one of the host implementations.
Severity	High
Status	Mitigated [4] [26]

For the parachain consensus it is essential that all honest validators can always agree on the validity of a parachain block/PVF. If the WASM code can somehow distinguish between the different host implementations this will allow creating a WASM blob which will accept a block with the Rust reference implementation and return an error when running in Kagome (or vice versa). This can be abused to create disputes between honest validators running different Polkadot host implementations.

There are currently several differences in the allocator behavior which can trigger this:

1. The header format in the host function memory allocator is incompatible between Kagome and Polkadot
2. No memory allocation size limit is configured in Kagome.
3. Kagome is allocating new memory before checking if there is a freed chunk that can be reused.
4. Kagome incorrectly uses the “__heap_base” global variable.

Attackers can trigger disputes between honest validators running Kagome and the reference implementation from Parity. In the foreseeable future it is likely that only a small fraction of the validators will run Kagome, so these validators will likely end up on the losing side of a dispute and get slashed.

Ensure that the behavior of the allocator is 100% by-bug compatible with the reference implementation. This involves making sure that the in-memory metadata format is 100% equal with the reference implementation and that an arbitrary

sequence of allocations/frees will result in exactly the same memory pointers being returned by the allocator.

To ensure that this condition will always hold we recommend creating a differential fuzzer which will generate a sequence of memory allocations/frees based on the fuzzer input and run these operations against both allocators.

5.16 Incorrect behavior in cryptographic batch verification

Attack scenario	**Disputes between honest host validators can be created**
Location	core/host_api/impl
Tracking	[27]
Attack impact	An attacker can submit a PVF which can distinguish between Kagome and Polkadot nodes, opening disputes after the execution fails on one of the host implementations.
Severity	High
Status	Mitigated [28]

The polkadot-sdk reference implementation still supports batched verification for compatibility reasons, even if no actual batching is taking place any more in current runtimes. The Kagome implementation has the batch verification disabled.

A malicious PVF can distinguish between the Kagome and the Polkadot-sdk host implementation, effectively triggering disputes (and slashing) of honest validators running different host implementations.

If Kagome is ever used with an old substrate-based runtime still using batch verification as originally intended and a majority of the validator set is running Kagome, this could lead to a situation where the runtime accepts invalid signatures, subsequently allowing stealing of user funds.

To mitigate this issue, Quadrivium must implement batch verification with exactly the same behavior as the implementation in polkadot-sdk.

5.17 Insecure TLS protocol versions enabled in SSL configurations

Attack scenario	**Kagome is prone to SSLv2 and SSLv3 vulnerabilities**
Location	telemetry/impl and offchain/impl
Tracking	[29]
Attack impact	An attacker could intercept and decrypt sensitive data
Severity	Medium
Status	Mitigated [30]

The SSL configuration allows the usage of SSLv2 and SSLv3 protocols, which are known to have significant vulnerabilities like POODLE and DROWN. The use of these old protocols undermines the security of encrypted channels, making them susceptible to interception, decryption, and manipulation by malicious actors and fails to comply with modern security standards.

Quadrivium should update the SSL configuration to only use secure and modern versions of the TLS protocol, specifically TLS 1.3.

5.18 Absence of secure-validator mode in Kagome

Attack scenario	**Kagome validators cannot use the secure mode which limits system and network exposure**
Location	core/parachain/pvf
Tracking	[31]
Attack impact	An attacker could exploit vulnerabilities in Kagome and have access to the underlying host system
Severity	Low
Status	Open

Polkadot implements a feature called “Secure-Validator Mode” which is designed to enhance the security of validators by restricting their operations and exposure to the network, minimizing the risk of malicious attacks and vulnerabilities. Kagome does not yet implement such a feature.

The absence of a secure mode in Kagome may raise users' concerns regarding the security posture and risk exposure of validators operating within the Kagome environment. Moreover, without a secure mode, Kagome validators may be more susceptible to a range of security threats, including but not limited to, direct attacks on the validator nodes, exploitation of vulnerabilities, and increased risk of participating in network consensuses under adversarial conditions.

To mitigate the risks posed by the absence of the Secure-Validator Mode in Kagome, we recommend mirroring the feature implementation of Polkadot in Kagome.

5.19 Inadequate memory clearing post-private key usage

Attack scenario	**Private key remains in the memory after usage**
Location	core/common, core/crypto
Tracking	[32]
Attack impact	An attacker could be able to extract the private key from memory by controlling another process on the same host
Severity	Low
Status	Mitigated [33]

The security assessment identified an issue where, after using the private key for signing operations, the memory allocated for storing the key is not adequately zeroed or cleared. This behavior leaves residual data in memory, which could potentially be accessed by unauthorized processes or users.

The primary risk associated with this issue is the potential exposure of sensitive private key information. Although classified as low risk due to the limited scenarios in which this vulnerability can be exploited, the persistence of private key data in memory after use can be a vector for targeted attacks aimed at extracting cryptographic keys.

To address this issue, it is recommended to implement the "OPENSSL_cleanse()" function from the OpenSSL library as this library is already being used within Kagome.

5.20 Buffer read overflow in core/host_api/impl/crypto_extension.cpp

Attack scenario	Potential buffer overflow
Location	core/host_api/impl
Tracking	[34]
Attack impact	An attacker might be able to access more data in the WASM memory than required
Severity	Low
Status	Mitigated [35]

The function "getMemory().loadN" is being used multiple times with a statically defined buffer size, rather than a size determined at runtime through "runtime::PtrSize". This static buffer size approach could potentially lead to buffer overflow issues.

If a load operation exceeds the bounds of the WASM memory, it does not lead to a system crash but results in an exception. However, if the load does not trigger an exception, the read operation extends beyond the intended buffer, accessing other data in the WASM memory. This data could then become available to the WASM code, potentially leading to unauthorized data exposure.

An untrusted user space buffer should always be checked that the expected size is correct.

6 Evolution suggestions

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

6.1 Address currently open security issues

We recommend addressing already known security issues in a timely manner to prevent attackers from exploiting them – even if an open issue has a limited impact, an attacker might use it as part of their exploitation chain, which may have a more severe impact on Kagome. To our knowledge all yet non-mitigated issues have already fixes in progress.

6.2 Further recommended best practices

Keep a manageable queue of *TODO* items. The codebase of Kagome has a high number of "TODO" comments, most of them without an assigned tracking issue. This could establish an incorrect development process, especially for new team members, and could lead to an unmanageable number of improvements or bug fixes that are incorrectly tracked, increasing the chances of vulnerabilities in the codebase. Our recommendation is to always track bugs, improvements and features correctly in GitHub, putting the link in the TODO in the source code, and approach them based on their criticality.

Implement extensive test cases. Kagome should implement at least the same test cases that Polkadot does. On top of this, any other Kagome specific corner case should be covered by tests. In our opinion, the current number of implemented test cases is not sufficient.

Properly comment the code, specifying reference implementation in Polkadot SDK and Polkadot Specifications. As Kagome is a node implementation of Polkadot, it should strictly follow the Polkadot Specifications and the Rust implementation. This also includes making Kagome bug-compatible with Polkadot on features that would/could break consensus. As of now, the code of Kagome has a different structure than the Polkadot SDK implementation, making it difficult to do a code comparison. Our recommendation is to include as many comments as possible on how the code is supposed to behave, and moreover, link the implementation to both the Polkadot SDK and Specifications.

Keep the project dependencies updated. Kagome is using the Hunter package manager to manage the dependencies of the project. The team should regularly check the latest versions of the packages that are being used in Kagome, and update them accordingly, accommodating any breaking changes.

Compile the Kagome node and tests with different sanitizers. A big part of the vulnerabilities that can be exploited in C++ projects are related to memory management. A good part of these issues could be discovered by compiling the Kagome node with different sanitizers (e.g.: address, threat, undefined, etc.). Such a node should then run for an extended period, eventually triggering crashes. The same also applies to the test cases, such as the Quadrivium team should run the test cases with different sanitizers. This process can be part of a CI/CD pipeline.

Regular code review and continuous fuzz testing. Regular code reviews are recommended to avoid introducing new logic bugs, while continuous fuzz testing can identify potential vulnerabilities early in the development process. Ideally, Quadrivium should continuously fuzz their code on each commit made to the codebase.

Create a differential fuzzer. As an alternative Polkadot node implementation, Kagome should always create the same output as Polkadot does. A differential fuzzer could discover edge cases where the two node implementations output a different result for the same input. Our recommendation is to develop differential fuzzers for different modules of the system. One example could be targeting the *"runtime_version"* host function.

Regular updates. New releases of Polkadot may contain fixes for critical security issues. Kagome should try to keep their implementation up to date with the latest Polkadot node version.

7 Bibliography

[1] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/28.

[2] [Online]. Available: https://github.com/qdrvm/kagome/pull/1985.

[3] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/17.

[4] [Online]. Available: https://github.com/qdrvm/kagome/pull/1908.

[5] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/26.

[6] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/25.

[7] [Online]. Available: https://github.com/libp2p/cpp-libp2p/pull/229.

[8] [Online]. Available: https://github.com/qdrvm/kagome/pull/1925.

[9] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/24.

[10] [Online]. Available: https://github.com/qdrvm/kagome/pull/1924.

[11] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/15.

[12] [Online]. Available: https://github.com/qdrvm/scale-codec-cpp/pull/21.

[13] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/34.

[14] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/30.

[15] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/32.

[16] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/31.

[17] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/27.

[18] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/23.

[19] [Online]. Available: https://github.com/qdrvm/kagome/pull/1964.

[20] [Online]. Available: https://github.com/qdrvm/kagome/pull/2009.

[21] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/21.

[22] [Online]. Available: https://github.com/qdrvm/kagome/pull/1946.

[23] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/18.

[24] [Online]. Available: https://github.com/qdrvm/kagome/pull/1907.

[25] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/16.

[26] [Online]. Available: https://github.com/qdrvm/kagome/pull/2063.

[27] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/19.

[28] [Online]. Available: https://github.com/qdrvm/kagome/pull/1906.

[29] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/20.

[30] [Online]. Available: https://github.com/qdrvm/kagome/pull/1905.

[31] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/33.

[32] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/29.

[33] [Online]. Available: https://github.com/qdrvm/kagome/pull/1997.

[34] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/22.

[35] [Online]. Available: https://github.com/qdrvm/KAGOME-audit/issues/22.

[36] [Online]. Available: https://github.com/paritytech/polkadot/tree/master/xcm/xcm-simulator/fuzzer.

[37] [Online]. Available: https://github.com/qdrvm/kagome/pull/1946.