Source Code Audit on Grandine Validator for Ethereum

Final Report and Management Summary

2024-11-29

CONFIDENTIAL

X41 D-Sec GmbH
Krefelder Str. 123
D-52070 Aachen
Amtsgericht Aachen: HRB19989

https://x41-dsec.de/
info@x41-dsec.de

Revision	*Date*	*Change*	*Author(s)*
1	2024-11-29	Final Report and Management Summary	MSc. H. Moesl-Canaval, MSc. A. Schloegl, BSc. C. Mayr

Contents

Dashboard

Target	
Customer	Ethereum
Name	Grandine
Type	Rust Application
Version	1.0.0.rc1
Engagement	
Type	Code Review
Consultants	3: MSc. H. Moesl-Canaval, MSc. A. Schloegl and BSc. C. Mayr
Engagement Effort	20 person-days, 2024-11-11 to 2024-11-22
Total issues found	0

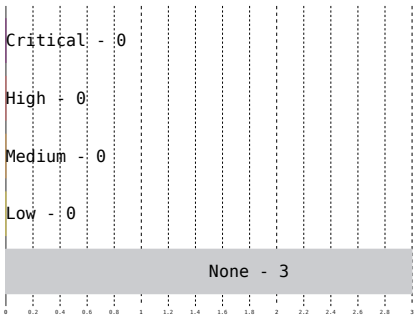

Figure 1: Issue Overview (l: Severity, r: CWE Distribution)

1 Executive Summary

In November 2024, X41 D-Sec GmbH performed a Code Review against the Grandine to identify vulnerabilities and weaknesses in the ecosystem.

Only three issues without a direct security impact were identified.

Grandine is a fast and lightweight Ethereum consensus client designed to handle the consensus layer of the Ethereum blockchain efficiently. It facilitates block validation, state synchronization, and participation in the consensus process while ensuring scalability and resource efficiency.

At the outset of the project, an initial kick-off meeting was held between X41 and the project maintainers to align on the engagement's scope. This meeting was instrumental in clarifying expectations for the assessment and identifying the key focus areas to prioritize.

Throughout the engagement, the testers maintained active communication with the library maintainers via a Signal group. The communication was outstanding, with prompt and helpful assistance provided whenever needed. Overall, the maintainers deserve high praise for their exceptional support and collaboration. It was a pleasure for the testing team to work alongside them.

As the code base itself is very large and Ethereum 2.0 is a rather complex stack, this audit focused on two areas of concern, as specified by the maintainers of Grandine:

- Vulnerabilities that cause Grandine clients to get slashed, and
- Denial-of-Service attack vectors.

Due to the extensive complexity of the Ethereum 2.0 specification and its substantial codebase, testers required a significant amount of time to fully familiarize themselves with the intricacies of the software ecosystem.

The test was performed by four experienced security experts between 2024-11-11 and 2024-11-22. In such an audit the testers inspect the source code statically for vulnerabilities, both on the implementation and on the design level.

The repository's structure is well-organized, and the coding style is consistently clean and pro-

fessional. However, the depth of certain call stacks can make it challenging to trace individual actions, potentially complicating the review process. Overall, the source code demonstrates a strong adherence to secure coding best practices, reflecting the developers' expertise in this area.

The codebase demonstrated strong adherence to safe Rust practices, with only minimal usage of the unsafe keyword. This minimized the risks associated with undefined behavior while leveraging Rust's guarantees for memory safety. Additionally, the absence of any Foreign Function Interface (FFI) usage further reinforced the code's security posture by eliminating potential vulnerabilities stemming from cross-language integrations.

Error handling was implemented consistently and effectively across the codebase, ensuring a high level of robustness. It was evident that the developers prioritized graceful degradation of functionality and maintained clear mechanisms for error propagation and resolution. This approach not only mitigated security-related risks but also improved the maintainability and debugability of the `Grandine` client.

The project exhibited a well-defined modular structure, with a strict separation of concerns across logical components. The use of Multi-Producer-Single-Consumer (mpsc) channels for inter-component communication enabled clean, thread-safe concurrency while maintaining simplicity and clarity in the design. This design choice minimized complexity and contributed positively to both maintainability and security.

Grandine does utilize external dependencies. Because of that, the security of the software project also depends on the security and robustness of those third party libraries. X41 recommends keeping all dependencies under a strict update regiment to plug any security issues that they contain in a timely fashion.

During the audit, it was observed that the Grandine client did not implement fuzz testing within its codebase. This omission represents a missed opportunity to proactively identify potential vulnerabilities stemming from unanticipated inputs or malformed data. Given the critical nature of the client's functionality, adopting fuzzing techniques is strongly recommended to improve its security posture. The implementation of fuzz testing would complement existing testing methodologies and provide additional assurance against exploitation of low-probability defects.

Fuzz testing is a highly effective security technique for uncovering vulnerabilities by generating and inputting randomized, unexpected, or malformed data into an application. It allows for the identification of edge cases, input validation issues, and other security flaws that might be overlooked during traditional testing methods. Integrating fuzz testing into the development pipeline can significantly enhance the robustness of an application and mitigate potential attack vectors, especially when paired with comprehensive unit tests and runtime monitoring.

Despite multiple auditors independently reviewing the same section of the code for better cov-

erage, no actual vulnerabilities having a direct security impact were identified during the review process.

Overall, the systems appear to be on a good security level and the choice to use Rust in the application's development is commendable, as it inherently bolsters security by mitigating common vulnerabilities.

To enhance the security posture further, it is recommended to implement additional security measures outlined in the Informational Notes. These notes highlight potential improvements, including addressing gaps in input validation, mitigating memory leaks, and resolving issues related to randomness.

It must be re-iterated that this assessment offered valuable insights into the security posture at the time of testing. However, it should be emphasized that no source code audit can definitively guarantee the absence of additional vulnerabilities within the software.

It is important to highlight that Rust, as a relatively new programming language, is still evolving rapidly. Changes to the Rust compiler, especially in its optimization logic, could inadvertently undermine the effectiveness of essential mitigations that are foundational to Rust's design and security guarantees.

All in all, given the use of the Grandine project, the code base would profit from recurring security audits, as changes within one part of the system can unintentionally impact the security of others.

2 Introduction

X41 reviewed Grandine, a high performance open-source Ethereum Consensus Layer client written in Rust.

An Ethereum Consensus Layer client interacts with various components of the blockchain network, making its security critical to ensure the integrity of the consensus mechanism and to protect the network and its participants from potential threats.

The confidentiality, integrity, and availability of data and functionality are important to maintaining trust and security within the Ethereum ecosystem. This is particularly sensitive in the context of preventing slashing validators using the `Grandine` client, as any compromise could lead to unjust penalties or financial losses.

Attackers may target an Ethereum Consensus Layer client, specifically the validator's functionality and slashing protection mechanisms, to exploit vulnerabilities that could lead to unjust slashing events. The security goals of the slashing mechanism are to disincentivize adversarial behavior and ensure robust detection of misconduct. A dishonest validator should not be able to act maliciously without detection or cause another innocent validator to be penalized. For example, if all slashers were ineffective during a given epoch, malicious activity might go unpunished, or an innocent validator might face consequences. Similarly, a malicious actor should not be able to manipulate a validator into committing a slashable offense, such as enticing a failover instance without adequate slashing protection, solely to claim whistleblower rewards. Such scenarios underscore the importance of safeguarding validators against attacks and ensuring the reliability of the slashing protection mechanisms.

2.1 Methodology

The review was based on a review of the source code.

A manual approach for penetration tests and for code reviews is used by X41. This process is

supported by tools such as static code analyzers and industry standard web application security tools[1].

X41 adheres to established standards for source code reviewing and penetration testing. These are in particular the *CERT Secure Coding*[2] standards and the *Study - A Penetration Testing Model*[3] of the German Federal Office for Information Security.

The workflow of source code reviews is shown in figure 2.1. In an initial, informal workshop regarding the design and architecture of the application a basic threat model is created. This is used to explore the source code for interesting attack surface and code paths. These are then audited manually and with the help of tools such as static analyzers and fuzzers. The identified issues are documented and can be used in a GAP analysis to highlight changes to previous audits.

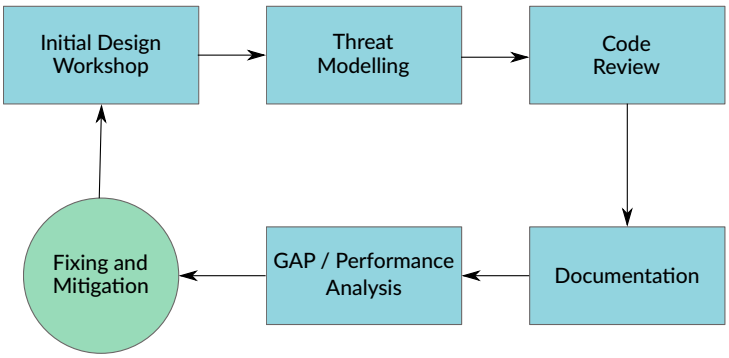

Figure 2.1: Code Review Methodology

[1] https://portswigger.net/burp
[2] https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards
[3] https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Penetration/penetration_pdf.pdf?__blob=publicationFile&v=1

2.2 Findings Overview

DESCRIPTION	SEVERITY	ID	REF
Cleartext Logging of Validator API Token to STDOUT	NONE	GR-CR-24-100	4.2.1
RSA Library Has Timing Side Channel on Private Keys	NONE	GR-CR-24-101	4.2.2
Inconsistent Verifier Implementations	NONE	GR-CR-24-102	4.2.3

Table 2.1: Security-Relevant Findings

2.3 Scope

The `Grandine` security review was performed on the following version:

`https://github.com/grandinetech/grandine/releases/tag/1.0.0.rc1`

During the initial kick-off meeting, the following key focus areas were highlighted:

- Slashing Protection:
  `https://github.com/grandinetech/grandine/tree/develop/slashing_protection`
- Built-in Validator:
  `https://github.com/grandinetech/grandine/tree/develop/validator`
- Doppelganger Protection:
  `https://github.com/grandinetech/grandine/tree/develop/doppelganger_protection`
- Signer:
  `https://github.com/grandinetech/grandine/tree/develop/signer`

Even though the listing above contains the primary focus points, the general goal of this source code audit was to identify issues that could compromise the validators functionality or integrity, such as vulnerabilities within the Slashing Protection mechanisms. Equally important were attacks that could lead DoS, which would disrupt validator operations.

2.4 Coverage

A security assessment attempts to find the most important or sometimes as many of the existing problems as possible, though it is practically never possible to rule out the possibility of additional weaknesses being found in the future.

The time allocated to X41 for this assessment was sufficient to yield a reasonable coverage of the given scope.

All parts of the scope were audited for unsafe management of memory and logic bugs and other common security issues.

Furthermore, the assessment covered the following areas, among others:

- `Grandine` is an Ethereum 2.0 consensus layer application. It can operate a beacon node, as well as an optional validator. This validator, along with its built-in slashing and doppelganger protection were audited in depth.
- Doppelganger protection is meant to prevent the accidental presence of two validators with the same set of keys, which incurs financial punishments in Ethereum. It accomplishes this by waiting for two full epochs to ensure none of the validator's keys are being used by another participant in the network. Attestations are only signed after a key has been observed as inactive by this best-effort protection method. The code audit raised no issues with the current implementation.
- Slashing protection is meant to prevent the validator from signing attestations that would lead to slashing. This is accomplished by refusing to sign a transaction if it conflicts with or surrounds a transaction that was already signed with this key, or vice versa. All checks outlined in the relevant specification are properly implemented, as confirmed in the code review. The review highlighted that the `Grandine` client consistently prioritizes caution in handling potential conflicts or slashing risks. This defensive programming approach contributes to the application's robust security posture.
- Slashing protection data can also be im-/exported using the interchange format. The used schema has been matched to that specified by the EIP[4], and serialization as well as deserialization have been audited and found free of vulnerabilities. Specifically, it was examined whether the `Grandine` client could potentially operate in the background during the import or export process, which might have introduced a race condition. It was confirmed that the underlying database layer ensures data integrity by preventing read or write operations during an ongoing database transaction for the export or import process.
- Various scenarios involving the potential loss of funds and slashing attacks on Ethereum consensus clients, as described in research papers and protocol specifications, were evaluated for applicability to the Grandine client during the pentest. Through static code analysis, none of these scenarios could be leveraged to compromise the funds of the client holder, indicating resilience against these types of threats.
- The signer component underwent a rigorous evaluation to identify potential cryptographic and logical vulnerabilities. It was determined that the implementation aligns with best practices and adheres to robust cryptographic principles, ensuring both integrity and security

[4] Ethereum Improvement Proposal

in the signing operations. No exploitable flaws were detected during the analysis.

- Additionally, the key store mechanisms were subjected to thorough scrutiny. The review confirmed that the implementation adheres to the EIP-2335 specification, ensuring compliance with established standards for key management. This alignment reflects a commitment to secure and reliable storage of cryptographic keys, enhancing the overall resilience of the system.
- The analysis of the validator API, which serves as an external-facing HTTP interface, revealed no exploitable DoS vectors or authentication weaknesses. This indicates a strong implementation of access control and input handling mechanisms.

2.5 Recommended Further Tests

X41 recommends continuing on the current path with regards to defensive programming style and architecture. Current strategies result in an impressive security posture, and the team should be commended.

For added security regular code audits, especially for newly added features should be undergone to ensure no new vulnerabilities arise.

3 Rating Methodology for Security Vulnerabilities

Security vulnerabilities are given a purely technical rating by the testers as they are discovered during the test. Business factors and financial risks for Ethereum are beyond the scope of a penetration test which focuses entirely on technical factors. Yet technical results from a penetration test may be an integral part of a general risk assessment. A penetration test is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

In total, five different ratings exist, which are as follows:

Severity Rating
None
Low
Medium
High
Critical

A low rating indicates that the vulnerability is either very hard for an attacker to exploit due to special circumstances, or that the impact of exploitation is limited, whereas findings with a medium rating are more likely to be exploited or have a higher impact. High and critical ratings are assigned when the testers deem the prerequisites realistic or trivial and the impact significant or very significant.

Findings with the rating 'none' are called informational findings and are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

Common Weakness Enumeration

The CWE[1] is a set of software weaknesses that allows the categorization of vulnerabilities and weaknesses in software. If applicable, X41 provides the CWE-ID for each vulnerability that is discovered during a test.

CWE is a very powerful method to categorize a vulnerability and to give general descriptions and solution advice on recurring vulnerability types. CWE is developed by *MITRE*[2]. More information can be found on the CWE website at `https://cwe.mitre.org/`.

[1] Common Weakness Enumeration
[2] `https://www.mitre.org`

4 Results

This chapter describes the results of this test. The security-relevant findings are documented in Section 4.1. Additionally, findings without a direct security impact are documented in Section 4.2.

4.1 Findings

During this assessment, no vulnerabilities were identified. This is in part due to the strict adherence to the specification, but also due to smart design choices, which limit the attack surface.

4.2 Informational Notes

The following observations do not have a direct security impact, but are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

4.2.1 GR-CR-24-100: Cleartext Logging of Validator API Token to STDOUT

Affected Component:	validator/src/api.rs:716

4.2.1.1 Description

The `Grandine` validator client provides users access to the validator API[1]. During a review of the REST[2] API, it was identified that the authorization token is being logged to STDOUT using the ***info()*** logging function.

While this issue does not present a direct exploitation vector, it introduces an unnecessary security risk. Logging sensitive information such as authorization tokens to STDOUT is generally unjustified and could inadvertently expose these tokens to entities not anticipated.

4.2.1.2 Solution Advice

X41 advises against logging credentials in clear text under any circumstances. If logging such information is absolutely necessary, it should be strictly limited to debug builds and handled with appropriate safeguards to prevent unintended exposure.

[1] Application Programming Interface
[2] Representational State Transfer

4.2.2 GR-CR-24-101: RSA Library Has Timing Side Channel on Private Keys

Affected Component: Cargo.toml

4.2.2.1 Description

While checking for outdated and vulnerable packages, it was noticed that the current rsa crate is vulnerable to a timing side-channel attack over the network.

The attack can be performed over the network, however, a practical attack would require immense preparation, as well as a large number of timed executions. So while the risk of the success is rather low, Rootsys would like to raise the awareness that the library should be updated as soon as a new version is available.

Inside the `grandine-1.0.0.rc1` project, run ***cargo audit***.

This will fetch the Rust security advisory database and check the contents of `Cargo.lock` against it.

```
cargo audit
[...]
Crate:     rsa
Version:   0.9.6
Title:     Marvin Attack: potential key recovery through timing sidechannels
Date:      2023-11-22
ID:        RUSTSEC-2023-0071
URL:       https://rustsec.org/advisories/RUSTSEC-2023-0071
Severity:  5.9 (medium)
Solution:  No fixed upgrade is available!
Dependency tree:
[...]

error: 1 vulnerability found!
warning: 5 allowed warnings found
```

Listing 4.1: Cargo Audit Output

4.2.2.2 Solution Advice

Notably, the testing team was unable to comprehensively prove any potential impact during the limited time frame granted for this review. As such, the wider implications remain unknown at

this point and should be subjected to internal research at the earliest possible convenience for the in-house team.

Generally speaking, the provision of robust supply chain security can be considerably challenging to provide to an optimal standard. Oftentimes, an easy or comprehensive solution cannot be offered, while the results and efficacy of the selected protection framework can vary depending on the integrated version of the deployed libraries.

Currently, no patched version of the rsa crate is available, however, X41 recommends regularly monitoring for such a release, and updating as soon as possible.

4.2.3 GR-CR-24-102: Inconsistent Verifier Implementations

Affected Component: helper_functions/src/verifier.rs:301

4.2.3.1 Description

Signature validation in the `Grandine` validator client is done using different implementations of the *`Verifier`* trait. During the review of the relevant code, it was noticed that actual verification for the *`MultiVerifier`* only happen when its ***finish*** method is called. By contrast, the *`SingleVerifier`* will verify signatures immediately upon aggregation.

While this issue does not present a direct exploitation vector, it introduces an unnecessary risk of failing to validate signatures when the incorrect implementation of *`Verifier`* is used.

4.2.3.2 Solution Advice

X41 recommends adding a state variable that tracks whether ***finish*** has been called on a given *`MultiVerifier`*. Combined with an implementation of the *`Drop`* trait throwing an error if ***finish*** was never called, this would catch logic errors during testing, should they ever arise.

5 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Source code audit of ISC BIND9 DNS server[1]
- Source code audit of the Git source code version control system[2]
- Review of the Mozilla Firefox updater[3]
- X41 Browser Security White Paper[4]
- Review of Cryptographic Protocols (Wire)[5]
- Identification of flaws in Fax Machines[6,7]
- Smartcard Stack Fuzzing[8]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

[1] `https://x41-dsec.de/news/security/research/source-code-audit/2024/02/13/bind9-security-audit/`
[2] `https://x41-dsec.de/security/research/news/2023/01/17/git-security-audit-ostif/`
[3] `https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/`
[4] `https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf`
[5] `https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf`
[6] `https://www.x41-dsec.de/lab/blog/fax/`
[7] `https://2018.zeronights.ru/en/reports/zero-fax-given/`
[8] `https://www.x41-dsec.de/lab/blog/smartcards/`

Acronyms

API Application Programming Interface 15
CWE Common Weakness Enumeration 13
REST Representational State Transfer 15
EIP Ethereum Improvement Proposal 10