HACKEN

RADIX SECURITY ANALYSIS

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Intro

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another party. Any subsequent publication of this report shall be without mandatory consent.

Name	Radix
Website	https://www.radixdlt.com/
Repository	https://github.com/radixdlt/radixdlt-scrypto
Commit	c1ca596b569720df93bfadb26e4bf7e2a1be6b54
Platform	L1
Network	Radix
Languages	Rust
Methodology	Blockchain Protocol and Security Analysis Methodology
Lead Auditor	Bartosz Barwikowski (b.barwikowski@hacken.io)
Approver	Luciano Ciattaglia (l.ciattaglia@hacken.io)
Timeline	14.08.2023 - 10.11.2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Summary

Radix is a layer-1 network explicitly designed for DeFi developers, dedicated to eliminating the barriers inhibiting the full potential of DeFi. It provides a unique smart contract environment centered on asset handling, along with a collection of pre-made and reusable DeFi "lego bricks". Radix's goal is to cultivate a self-sustaining developer ecosystem with its Developer Royalty System, offering incentives for developers contributing to the network. By implementing its unique Cerberus consensus algorithm and harnessing parallel processing, Radix achieves infinite scalability and seamless interoperability between dApps.

The core focus of the audit is the Radix Engine, the main execution component of the Radix layer-1 network. Built on WASM VM, it offers a specialized environment for running DeFi-centric Scrypto applications. The engine uses well-defined finite state machines (FSMs) to control tokens and other assets, guaranteeing secure and predictable DeFi transactions. With the introduction of Radix Engine v2, "smart contract" programmability was brought to the platform through Scrypto, fostering asset-oriented, FSM-based financial assets and dApps development. In 2023, the engine's WASM interface received a substantial update, providing a more refined low-level WASM API and improving its Scrypto function export signatures. This revision simplified Scrypto contract execution, aligning with the network's objective of making DeFi dApp development more efficient and secure.

Scrypto is an asset-focused smart contract language employed within the Radix Engine. It empowers developers to effortlessly create and engage with assets on the Radix platform, promoting a safer and more practical DeFi dApp development process. Leveraging Scrypto, developers can concentrate on their business logic, entrusting the Radix Engine with asset management and authorization.

Documentation quality

Radix Engine offers a robust set of developer documentation enriched with illustrative example projects. The source code is meticulously crafted, self-explanatory, and well-written, considerably easing developers' grasp of its nuances.

The system is complemented with a comprehensive assortment of native blueprints and components. These are well-documented and straightforward, facilitating a seamless integration experience for developers. Radix Engine has prioritized clarity and user-friendliness in these critical areas, ensuring developers have the resources they need to excel.

Nevertheless, the low-level VM API is a minor area where Radix's documentation could use further refinement. It's important to note that this API is seldom accessed by most users and developers. And even when accessed, the intelligently constructed source code generally provides ample clarity. However, for an all-encompassing documentation suite, a brief elaboration on this segment would add to the completeness.

While the typical user or developer might not engage with the low-level VM API, ensuring thoroughness in every documentation aspect amplifies the engine's perceived comprehensiveness. To reach the pinnacle of documentation perfection, Radix should consider a slight enhancement in this particular section.

The total Documentation Quality score is **10** out of 10.

Code quality

Radix Engine codebase presents a high standard of craftsmanship, boasting well-written, largely self-explanatory code. The project's structure is well organized, making it relatively straightforward to understand and engage with. It is replete with a variety of tests, including fuzz tests, and incorporates developer-friendly tools for efficient working and debugging. Furthermore, it adheres to good coding practices in Rust.

The latest changes implemented a new error-handling mechanism, making many previously critical issues insignificant, which highly improved project stability.

The total Code Quality score is **10** out of 10.

Architecture quality

The architecture of the Radix Engine is well-conceived and executed, utilizing a contemporary approach for the VM implementation and fostering an impressive ecosystem for users and developers alike. This approach has the potential to greatly enhance the process of "smart contract" development, making it both easier and more secure, even outperforming the Solidity equivalent in user-friendliness.

However, there are a few minor issues that need to be addressed. Currently, the size of packages ("smart contracts") is rather large, and in many instances, their execution can be relatively slow. These are areas that could be improved to enhance the overall performance and efficiency of the system. While these issues are certainly noteworthy, they are present in most similar projects and are surmountable, and with time and consistent development efforts, they should be rectified. This will further solidify the architecture quality of the Radix Engine.

The architecture quality score is **10** out of 10.

Security score

During the second security audit of Radix, one critical issue and two low-severity issues were uncovered. The issues were confirmed and fixed by the radix team.

The security score is **10** out of 10.

Total score

Considering all metrics, the total score of the report is **10.0** out of 10.

Findings count and definitions

Severity	Findings
Critical	1
High	0
Medium	0
Low	2
Total	3

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Scope of the audit

Protocol Audit

Native blueprints

- Account + Access Controller
- Resource Manager + Vaults + Buckets + Proofs
- Package + Transaction Processor
- Consensus Manager + Validator

Authentication

- Auth implementation review
- Attack scenarios analysis (permission escalation, auth bypass ,...)

Costing and Limit Models

- Costing Implementation review
- Limits Implementation review
- Attack scenarios analysis (liveness, finality, eclipse, double spend, ...)

VM Engine

- VM implementatio nreview (instructions, state transition, ...)
- Common VM Vulnerabilities review
- Attack scenarios analysis (Gas, race conditions, stack, DoS, state implosion, ...)

Protocol Tests

VM Tests

- Environment Setup
- Fuzz tests

Fuzz Tests

- Focused on resource invariants

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Issues

Lack of 2nd Resource Address Validation

Multiple instances within the `resource` blueprints allow for interactions between two resources without confirming their identical type. A prime example is the `put` function in the fungible vault (`resource/blueprints/fungible/fungible_vault.rs`), which omits a crucial check to ensure the Bucket's resource type aligns with the vault's.

ID	RDX-100
Scope	Native blueprint
Severity	CRITICAL
Vulnerability Type	Insufficient Verification of Data Authenticity (CWE-345)
Status	Fixed

Description

In the implementation of the `put` function:

```
pub fn put<Y>(bucket: Bucket, api: &mut Y) -> Result<(), RuntimeError>
where
    Y: ClientApi<RuntimeError>,
{
    Self::assert_not_frozen(VaultFreezeFlags::DEPOSIT, api)?;
    // Drop other bucket
    let other_bucket = drop_fungible_bucket(bucket.0.as_node_id(), api)?;
    let amount = other_bucket.liquid.amount();
    // Put
    Self::internal_put(other_bucket.liquid, api)?;
    Runtime::emit_event(api, events::fungible_vault::DepositEvent { amount })?;
    Ok(())
}
```

The main issue lies in the absence of a validation step to verify the resource address of the Bucket against the vault's address. This gap allows for the deposit of Buckets with differing token types into a vault, such as inserting a non-XRD token into an XRD (native radix token) vault. This vulnerability enables the conversion of one token type to another, potentially leading to unrestricted generation of new XRD tokens.

This flaw is not limited to the fungible vault; it also affects non-fungible vaults and bucket implementations.

Recommendation

To mitigate such vulnerabilities, it is vital to incorporate stringent validation processes in scenarios where resources interact. These checks must confirm the identical nature of resource types involved in any transaction or interaction. Furthermore, implementing comprehensive logging and monitoring mechanisms for such interactions can provide early detection of anomalous activities, enhancing overall system security and integrity.

Overflow in `compare_current_time` Function

The `compare_current_time` function in the consensus manager exhibits two instances of overflow.

ID	RDX-101
Scope	Native blueprint
Severity	**LOW**
Vulnerability Type	Integer Overflow or Wraparound (CWE-190)
Status	Fixed

Description

The `compare_current_time` function, located in `blueprints/consensus_manager/consensus_manager.rs`, is implemented as follows:

```
pub fn put<Y>(bucket: Bucket, api: &mut Y) -> Result<(), RuntimeError>
where
    Y: ClientApi<RuntimeError>,
{
    pub(crate) fn compare_current_time<Y>(
        other_arbitrary_precision_instant: Instant,
        precision: TimePrecision,
        operator: TimeComparisonOperator,
        api: &mut Y,
    ) -> Result<bool, RuntimeError>
    where
        Y: ClientApi<RuntimeError>,
    {
        match precision {
            TimePrecision::Minute => {
                let other_epoch_minute = Self::milli_to_minute(
                    other_arbitrary_precision_instant.seconds_since_unix_epoch * MILLIS_IN_SECOND,
                );
                let handle = api.actor_open_field(
                    ACTOR_STATE_SELF,
                    ConsensusManagerField::ProposerMinuteTimestamp.into(),
                    LockFlags::read_only(),
                )?;
                let proposer_minute_timestamp = api
                    .field_read_typed::<ConsensusManagerProposerMinuteTimestampFieldPayload>(
                        handle,
                    )?
                    .into_latest();
                api.field_close(handle)?;
                // convert back to Instant only for comparison operation
                let proposer_instant =
                    Self::epoch_minute_to_instant(proposer_minute_timestamp.epoch_minute);
                let other_instant = Self::epoch_minute_to_instant(other_epoch_minute);
                let result = proposer_instant.compare(other_instant, operator);
                Ok(result)
            }
        }
    }
}
```

The primary issue arises in the expression `other_arbitrary_precision_instant.seconds_since_unix_epoch * MILLIS_IN_SECOND`, which can lead to an overflow with excessively large values (e.g., i64::MAX). Additionally, the `milli_to_minute` function can trigger a panic due to the `unwrap` call when `epoch_milli` is 1000 times larger than i32::MAX or i32::MIN. The Radix team has implemented a mechanism to catch panics in native blueprints, which prevents application crashes but causes transactions to fail.

Recommendation

A thorough validation should be performed to ensure `other_arbitrary_precision_instant.seconds_since_unix_epoch` is neither negative nor exceeds i32::MAX. Implementing safeguards against overflow in calculations, such as using checked arithmetic functions (`checked_mul`, `checked_add`, etc.), can prevent such vulnerabilities. Additionally, enhancing error handling in functions like `milli_to_minute` to gracefully handle edge cases instead of using `unwrap` will improve the robustness and security of the system.

Potential Supply Chain Attack in `scrypto` Dependency Management

The dependency management in `scrypto`-based packages currently poses a risk for a supply chain attack due to the use of over 100 third-party packages without specified versions.

ID	RDX-102
Scope	Dependencies
Severity	**LOW**
Vulnerability Type	Inclusion of Functionality from Untrusted Control Sphere (CWE-829)
Status	Fixed

Description

The primary concern stems from the absence of a `cargo.lock` file in the project, which is crucial for consistent dependency management. By default, packages built using `scrypto` incorporate more than 100 third-party packages. Without version locking, there is a theoretical risk that any of these dependencies could be compromised with malicious code. Such a compromise might go unnoticed for a significant period, potentially undermining the security and integrity of the packages developed using this framework.

Recommendation

To address this vulnerability, implementing a `cargo.lock` file in the project is essential. This file will lock all dependencies to specific, verified versions, greatly reducing the risk of a supply chain attack. Furthermore, actively monitoring the dependencies for security updates and known vulnerabilities is crucial. Keeping the dependencies updated and secure requires a proactive approach, including subscribing to security bulletins and using automated tools for vulnerability scanning. This strategy will ensure the reliability and security of the dependencies and, by extension, the integrity of the final product.

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Tests and code coverage

As of commit `d16b3ffa65fea417e9c6d01d76456a0b36c060fa` the project has a total of 4237 tests. Additionally, the project contains multiple fuzz target and error injection tests.

The project has the following code coverage:

Module	Regions (Covered/Total/%)	Functions (Covered/Total/%)	Lines (Covered/Total/%)
monkey-tests	437 / 543 / 80.48%	158 / 223 / 70.85%	1526 / 1631 / 93.56%
native-sdk	648 / 834 / 77.70%	124 / 158 / 78.48%	1727 / 2117 / 81.58%
radix-engine-common	31620 / 33027 / 95.74%	2080 / 2578 / 80.68%	10896 / 11908 / 91.50%
radix-engine-derive	106 / 124 / 85.48%	67 / 78 / 85.90%	689 / 709 / 97.18%
radix-engine-interface	5338 / 8962 / 59.56%	1957 / 3080 / 63.54%	4165 / 5578 / 74.67%
radix-engine-macros	108 / 144 / 75.00%	17 / 23 / 73.91%	159 / 214 / 74.30%
radix-engine-profiling	1 / 1 / 100.00%	1 / 1 / 100.00%	3 / 3 / 100.00%
radix-engine-queries	442 / 690 / 64.06%	84 / 135 / 62.22%	759 / 998 / 76.05%
radix-engine-store-interface	120 / 191 / 62.83%	62 / 90 / 68.89%	202 / 254 / 79.53%
radix-engine-stores	1003 / 1338 / 74.96%	390 / 527 / 74.00%	2267 / 2651 / 85.51%
radix-engine	23231 / 31406 / 73.97%	4040 / 5522 / 73.16%	44363 / 50493 / 87.86%
sbor-derive-common	833 / 1006 / 82.80%	121 / 123 / 98.37%	2155 / 2257 / 95.48%
sbor-derive	30 / 48 / 62.50%	20 / 34 / 58.82%	76 / 115 / 66.09%
sbor-tests	6 / 12 / 50.00%	5 / 10 / 50.00%	16 / 32 / 50.00%
sbor	4691 / 6644 / 70.61%	917 / 1185 / 77.38%	8203 / 9641 / 85.08%
scrypto-derive	768 / 1069 / 71.84%	70 / 85 / 82.35%	2474 / 2856 / 86.62%
scrypto-schema	324 / 537 / 60.34%	108 / 148 / 72.97%	190 / 253 / 75.10%
scrypto-test	392 / 585 / 67.01%	156 / 229 / 68.12%	1589 / 2186 / 72.69%
scrypto-unit	497 / 594 / 83.67%	243 / 290 / 83.79%	2897 / 3318 / 87.31%
transaction-scenarios	939 / 1134 / 82.80%	340 / 363 / 93.66%	3407 / 3496 / 97.45%
transaction	55917 / 58282 / 95.94%	1402 / 2549 / 55.00%	8512 / 9772 / 87.11%
utils	37 / 43 / 86.05%	33 / 38 / 86.84%	139 / 152 / 91.45%

The most important part of the project, `radix-engine` has the following code coverage:

Module	Regions (Covered/Total/%)	Functions (Covered/Total/%)	Lines (Covered/Total/%)
blueprints	10393 / 14316 / 72.60%	1432 / 1994 / 71.82%	19221 / 21524 / 89.30%

kernel	1867 / 2498 / 74.74%	409 / 567 / 72.13%	3268 / 3692 / 88.52%
system	6093 / 7860 / 77.52%	1119 / 1476 / 75.81%	12308 / 14055 / 87.57%
track	636 / 770 / 82.60%	151 / 195 / 77.44%	1477 / 1594 / 92.66%
transaction	1176 / 1656 / 71.01%	274 / 391 / 70.08%	2433 / 2837 / 85.76%
utils	182 / 284 / 64.08%	26 / 40 / 65.00%	331 / 517 / 64.02%
vm	2286 / 2903 / 78.75%	496 / 632 / 78.48%	4917 / 5523 / 89.03%

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Disclaimers

Hacken disclaimer

The code base provided for audit has been analyzed according to the latest industry code quality, software processes and cybersecurity practices at the date of this report, with discovered security vulnerabilities and issues the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functional specifications). The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code (branch/tag/commit hash) submitted to and reviewed, so it may not be relevant to any other branch. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits, public bug bounty program and CI/CD process to ensure security and code quality. English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical disclaimer

Protocol Level Systems are deployed and executed on hardware and software underlying platforms and platform dependencies (Operating System, System Libraries, Runtime Virtual Machines, linked libraries, etc.). The platform, programming languages, and other software related to the Protocol Level System may have vulnerabilities that can lead to security issues and exploits. Thus, Consultant cannot guarantee the explicit security of the Protocol system in full execution environment stack (hardware, OS, libraries, etc.)