Code Review of the Rust VMM Project
for The Rust VMM project

Final Report and Management Summary

2023-11-07

PUBLIC

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

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

Organized by the Open Source Technology Improvement Fund

Revision	*Date*	*Change*	*Author(s)*
1	2023-09-29	Final Report and Management Summary	M.Sc. H. Moesl-Canaval, R. Femmer

Contents

Dashboard

Target	
Customer	The Rust VMM project
Name	RustVMM
Type	Virtual Machine Monitor
Version	As indicated in Scope
Engagement	
Type	Code Audit
Consultants	2: H. Moesl-Canaval (M.Sc.) and R. Femmer
Engagement Effort	30 person-days, 2023-09-11 to 2023-09-29
Total issues found	0

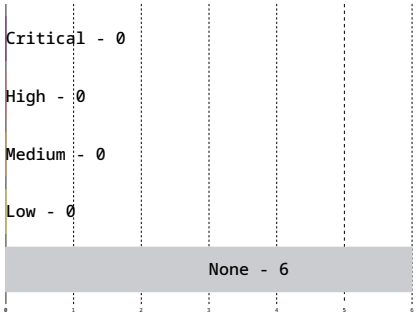

Figure 1: Issue Overview (Severity)

1 Executive Summary

In September 2023, X41 D-Sec GmbH performed a Code Audit against the RustVMM to identify vulnerabilities and weaknesses in the ecosystem.

Only six issues without a direct security impact were identified.

RustVMM provides a collection of crates that can be used to implement a Virtual Machine Monitor (VMM), which can manage a number of virtual machines. The management includes starting and stopping the machines and providing resources to them like virtualized devices (network, storage), memory and CPUs. Vulnerabilities in the application would allow an attacker to execute code on the host machine or other guest machines, or break the isolation by disclosing data belonging to the host machine or other guest machines. Further, due to resource management being an integral part of the VMM, resource exhaustion scenarios pose a considerable risk to the operator of the VMM.

At the beginning of the project, an initial kick-off meeting was set up between X41, OSTIF and the maintainers of the project in order to align on the scope of this engagement. The meeting helped to clarify the expectations of this assessment and also narrowed down the key focus areas.

Throughout the engagement the testers were in communication with OSTIF and the maintainers of the library through a public Slack channel and email. The communication was excellent, and help was provided whenever requested. Generally speaking, OSTIF as well as the maintainers of the libraries deserve a lot of praise for their overall support and assistance. It was a pleasure for the testing team working with them.

The test was performed by two experienced security experts between 2023-09-11 and 2023-09-29. In such an audit the testers inspect the source code statically for vulnerabilities, both on the implementation and on the design level.

Generally, the code quality is very high. Components are divided into sensible units that allow to assess their security in isolation. It is obvious that security was a main concern during the design and implementation of `rust-vmm`. While a lot of the security guarantees are provided by the hypervisor, `rust-vmm` does an excellent job at all other parts of the system. Because

of this no issues with direct security impact could be identified. Only minor issues that may become security problems in the future could be documented as informational notes. The most severe one, that may have lead to possible memory corruption at some point, was reported to the `rust-vmm` maintainers and fixed within a week, which is another testament to the overall security posture of the project.

The `rust-vmm` does not have many dependencies, however 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.

Stakeholders using `rust-vmm` components may also consider to audit the underlying hypervisor.

Despite multiple auditors independently reviewing the same section of the code for better coverage, no actual vulnerabilities having a direct security impact were identified during the review process.

It is worth emphasizing that the `rust-vmm` project was put through rigorous testing by X41's team and, overall, it was found to be highly robust and secure. The testing process revealed that all Rust crates involved in the project are designed and implemented with great care and attention to detail, and it demonstrated a strong ability to withstand scrutiny. As a result, the overall impression and outcome of this security assessment is very positive.

The `rust-vmm`, as indicated by its name, is developed in Rust—a language renowned for its built-in memory management features. Because of that, Rust has emerged as a favored option for developers keen on circumventing traditional memory corruption vulnerabilities and race conditions, pitfalls commonly associated with memory-unsafe languages like C/C++. This inherent security benefit, paired with Rust's robust programming capabilities, underscores its growing appeal in contemporary software development.

To further improve the security posture, it is encouraged to implement additional security controls, which have been listed as part of the Informational Notes list. These describe potential improvements with regards to missing input validation checks, memory leak mitigation and randomness issues.

Again, it must be reiterated that this assessment provided valuable insights into the security posture at the time of testing, but it is important to note that any source code audit is unable to guarantee that the software complex is free of additional bugs.

Finally, it is important to emphasize that Rust is a relatively new programming language and, as such, is undergoing significant development. Alterations to the Rust compiler, particularly in the realm of optimization logic, have the potential to compromise the efficacy of key mitigations intrinsic to the Rust programming language.

All in all, given the in-production and high-profile use of the `rust-vmm` project, the code base would profit from recurring security audits as changes within one part of the system may have unintentional security impact to other parts.

2 Introduction

X41 reviewed the `rust-vmm` ecosystem, which provides a set of components to build custom Virtual Machine Monitors (VMM). The packages - also called crates - are in use in various projects, among them `CrosVM` and `Firecracker`.

A Virtual Machine Monitor exposes various components to untrusted code and therefore the security of the VMM is critical in order to not expose infrastructure or the users of the infrastructure to malicious actors.

Attackers might attempt to target the memory management system and virtualized devices that offer functionality to virtual machines, with the aim of executing code or revealing confidential information from the host system or other guest systems.

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

[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`

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.

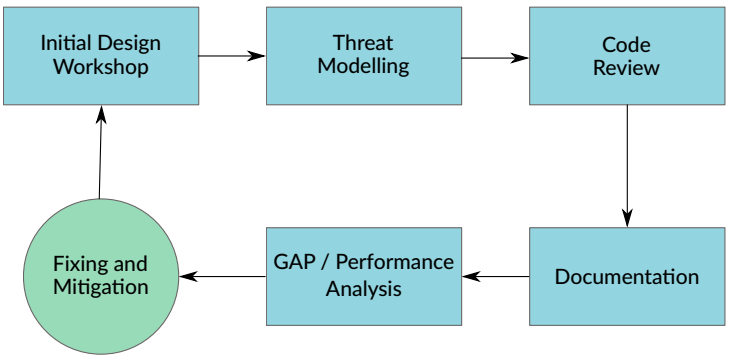

Figure 2.1: Code Review Methodology

2.2 Findings Overview

DESCRIPTION	SEVERITY	ID	REF
Random Module not Random	NONE	RVMM-CR-23-100	4.2.1
Missing Divide-by-Zero Check	NONE	RVMM-CR-23-101	4.2.2
Missing Input Parameter Validation	NONE	RVMM-CR-23-102	4.2.3
Possible Out-of-Bounds Write in FamStruct	NONE	RVMM-CR-23-103	4.2.4
Possible Memory Leak in Temporary File Creation Routine	NONE	RVMM-CR-23-104	4.2.5
GuestMemory::try_access() Invariant not Checked	NONE	RVMM-CR-23-105	4.2.6

Table 2.1: Security-Relevant Findings

2.3 Scope

The `rust-vmm` security review was performed on the following Rust crates:

- kvm-ioctls
  - Commit: e4ee241cc188f253c184ed48b3d5c4fb58a86da9
- linux-loader
  - Commit: ddb20723b8ac0119676b53afe09401da1e72d233
- seccompiler
  - Commit: 184f2d6838b753bcdf09be7c6b46934f33c86535
- vhost
  - Commit: 3808f9d0032bf763d79ff94504ec756f9065c83a
- vhost-device
  - Commit: 38caab24c5087551d6ec12d0002df798e5e4f5ac
- vm-allocator
  - Commit: 20983a10b01549c00e1e0772fe8e2ccf07815cbf
- vm-device
  - Commit: 39ceae64edc11860c44679f4b1563cec358047e2
- vmm-sys-util

  - Commit: 3fb8f7693228ae3e582228414a0daa1b90224c84

- vm-memory
  - Commit: aff1dd4a5259f7deba56692840f7a2d9ca34c9c8
- vm-superio
  - Commit: 0a3ae7fa5957098371ce934b476f729621fb8d07
- vm-virtio

As part of this source code audit, various trust boundaries within the virtualized environment were examined. These boundaries encompassed the distinctions of Guest-Host, Guest-Guest, Guest (Ring 3) to Guest (Ring 0), with an added consideration for Guest (Ring 3) to Guest (Ring -1) attacks. Part of the threat model were vulnerabilities such as information disclosure, memory corruption, and Denial of Service (DoS) attacks that might exploit these trust boundaries. Additionally, the testing team focused on logical issues within the allotted time.

Within the scope of the source code audit, only the most recent release of any given crate was reviewed. Therefore, the commits or branches that were of primary interest pertained to the HEAD of the main branch, which usually corresponds closely to the most recent release.

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.

Suggestions for next steps in securing this scope can be found in section 2.5.

2.5 Recommended Further Tests

Fuzz testing is, in general, essential for the overall security of the `rust-vmm` project. It is commendable that the `vm-virtio` crate has already incorporated fuzz testing methodologies. As the

project progresses, it should be evaluated whether other crates of the `rust-vmm` project could benefit from fuzz testing.

X41 recommends to mitigate the issues described in this report.

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 The Rust VMM project 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.

3.1 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

No issues with a direct impact on the security were identified.

During the code audit of the `rust-vmm` project, no noteworthy issues were uncovered that could potentially result in vulnerabilities.

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 RVMM-CR-23-100: Random Module not Random

Affected Component:	vmm-sys-util/src/rand.rs

4.2.1.1 Description

The *vmm-sys-util* crate exposes a module called *rand* which exposing ***rand_alphanumerics()*** and ***rand_bytes()***. As their name suggests, these functions generate random sequences of *OsString* and *u8* values, however, they rely on the *RDTSC* for *x86_64* systems and libc's ***clock_gettime()*** for randomness. Frequently, people assume that bytes or sequences labeled as random are unpredictable and might use them for security-sensitive purposes. This could pose a problem in specific situations.

We found no relevant code in rust-vmm that would present itself as problematic, however, the ***rand_alphanumerics()*** is already being used to construct temporary files on Windows (c.f. `vmm-sys-util/src/tempfile.rs`), which are usually expected to be unpredictable.

4.2.1.2 Solution Advice

X41 recommends to use, and if needed, wrap the *rand* or any other hardened crate providing a better source of randomness.

4.2.2 RVMM-CR-23-101: Missing Divide-by-Zero Check

Affected Component:	vm-memory/src/bitmap/backend/atomic_bitmap.rs

4.2.2.1 Description

While reviewing the `vm-memory` Rust crate, it was noticed that the function ***AtomicBitmap::new()*** does not properly check the *`page_size`* parameter for being zero. As this parameter is being used as a divisor within the function, a value of zero leads to a Divide-by-zero CPU fault resulting in an application panic. Listing 4.1 shows the affected code snippet.

While the function is solely used within the `rust-vmm` ecosystem with valid `page_size` values unequal to zero, the function is public and might be used to construct a bitmap by other 3rd-party-crates using `vm-memory`.

```
impl AtomicBitmap {
    /// Create a new bitmap of `byte_size`, with one bit per page. This is effectively
    /// rounded up, and we get a new vector of the next multiple of 64 bigger than `bit_size`.
    pub fn new(byte_size: usize, page_size: usize) -> Self {
        let mut num_pages = byte_size / page_size;
        if byte_size % page_size > 0 {
            num_pages += 1;
        }

        // Adding one entry element more just in case `num_pages` is not a multiple of `64`.
        let map_size = num_pages / 64 + 1;
        let map: Vec<AtomicU64> = (0..map_size).map(|_| AtomicU64::new(0)).collect();

        AtomicBitmap {
            map,
            size: num_pages,
            page_size,
        }
    }

    [...]
}
```

Listing 4.1: AtomicBitmap::new() Function

4.2.2.2 Solution Advice

X41 advises performing thorough input parameter validation to prevent any possibility of encountering a Divide-by-zero situation.

4.2.3 RVMM-CR-23-102: Missing Input Parameter Validation

Affected Component:	vhost/crates/vhost-user-backend/src/handler.rs

4.2.3.1 Description

While reviewing the `vhost` crate, it was found that the function ***VHostUserHandler::set_vring_base()*** lacks validation of the input parameter `index`, which is used for indexing an internal vector and could potentially lead to a Out-of-Bounds write resulting in an application panic. Listing 4.2 depicts the vulnerable code segment.

```
impl<S, V, B> VhostUserBackendReqHandlerMut for VhostUserHandler<S, V, B>
where
    S: VhostUserBackend<V, B>,
    V: VringT<GM<B>>,
    B: NewBitmap + Clone,
{

    [...]

    fn set_vring_base(&mut self, index: u32, base: u32) -> VhostUserResult<()> {
        let event_idx: bool = (self.acked_features & (1 << VIRTIO_RING_F_EVENT_IDX)) != 0;

        self.vrings[index as usize].set_queue_next_avail(base as u16);
        self.vrings[index as usize].set_queue_event_idx(event_idx);
        self.backend.set_event_idx(event_idx);

        Ok(())
    }
}
```

Listing 4.2: VHostUserHandler::set_vring_base() Function

4.2.3.2 Solution Advice

X41 advises performing thorough input parameter validation to prevent any possibility of encountering potential Out-of-Bounds read/writes.

4.2.4 RVMM-CR-23-103: Possible Out-of-Bounds Write in FamStruct

Affected Component: vmm-sys-util/src/fam.rs

4.2.4.1 Description

X41 found that there is a possible Out-of-bounds write in ***FamStruct::set_len()***, if the argument *len* holds values that - when cast from it's original type *usize* to *isize* - become negative, the code will proceed to zero out *len* bytes of memory, which in all likelihood will crash the process. The provided test in listing 4.3 can trigger this issue.

```
generate_fam_struct_impl!(MockFamStructU8, u8, entries, u32, len, 100);
type MockFamStructWrapperU8 = FamStructWrapper<MockFamStructU8>;
#[test]
fn test_invalid_type_conversion() {
    let mut adapter = MockFamStructWrapperU8::new(10).unwrap();
    assert!(matches!(
        adapter.set_len(0xffff_ffff_ffff_ff00),
        Err(Error::SizeLimitExceeded)
    ));
}
```

Listing 4.3: Test Triggering a SEGFAULT Due to Out-of-Bounds Write

Instead of the expected error of an exceeded size limit, the process crashes. The security impact is negligible since the method is not exposed to the user of the *vmm-sys-util* crate. However, the maintainers may decide to expose the method at some point in the future, which could lead to a security issue.

4.2.4.2 Solution Advice

X41 disclosed this possible memory corruption to the maintainers prior to the release of this report. A fix [1] was committed to vmm-sys-util within a week.

[1] https://github.com/rust-vmm/vmm-sys-util/commit/5bf1061dd9fc18a2d25eda12ce2d2ea63fb999d4

4.2.5 RVMM-CR-23-104: Possible Memory Leak in Temporary File Creation Routine

Affected Component:	vmm-sys-util/src/tempfile.rs

4.2.5.1 Description

While reviewing the `vmm-sys-util` Rust crate, it was noticed that on a very rare occasion the memory region of a CString depicting a temporary file name does not get freed after usage resulting in a memory leak situation.

According to the Rust documentation, a call to ***std::ffi::CString::into_raw()***, which transfers ownership of the string to a C caller, must be accompanied by a final call to ***std::ffi::CString::from_raw()*** which retakes the ownership again. Failure in doing so results in a memory leak.

In the code snippet shown in listing 4.4, ownership of a CString is transferred in (1) through calling ***into_raw()***to C and returned in (3) via ***from_raw()*** to Rust. In case of an error in `mkstemp()` in (2), the function is exited without calling ***from_raw()*** leading to the memory leak. According to the documentation of ***mkstemp()*** (https://man7.org/linux/man-pages/man3/mkstemp.3.html) this can only happen if there already exists a file which happens to have the same file name as the temporary file.

```
1       pub fn new_with_prefix<P: AsRef<OsStr>>(prefix: P) -> Result<TempFile> {
2           use std::ffi::CString;
3           use std::os::unix::{ffi::OsStrExt, io::FromRawFd};
4
5           let mut os_fname = prefix.as_ref().to_os_string();
6           os_fname.push("XXXXXX");
7
8           let raw_fname = match CString::new(os_fname.as_bytes()) {
9       (1)     Ok(c_string) => c_string.into_raw(),
10              Err(_) => return Err(Error::new(libc::EINVAL)),
11          };
12
13          // SAFETY: Safe because `raw_fname` originates from CString::into_raw, meaning
14          // it is a pointer to a nul-terminated sequence of characters.
15          let fd = unsafe { libc::mkstemp(raw_fname) };
16          if fd == -1 {
17      (2)     return errno_result();
18          }
19
20          // SAFETY: raw_fname originates from a call to CString::into_raw. The length
21          // of the string has not changed, as mkstemp returns a valid file name, and
22          // '\0' cannot be part of a valid filename.
```

```
23      (3) let c_tempname = unsafe { CString::from_raw(raw_fname) };
24          let os_tempname = OsStr::from_bytes(c_tempname.as_bytes());
25
26          // SAFETY: Safe because we checked `fd != -1` above and we uniquely own the file
27          // descriptor. This `fd` will be freed etc when `File` and thus
28          // `TempFile` goes out of scope.
29          let file = unsafe { File::from_raw_fd(fd) };
30
31          Ok(TempFile {
32              path: PathBuf::from(os_tempname),
33              file: Some(file),
34          })
35      }
```

Listing 4.4: TempFile::new_with_prefix() Function

4.2.5.2 Solution Advice

X41 advises calling ***std::ffi::CString::from_raw()*** before returning from the function to avoid memory leaks. This also affects other types such as `std::ffi::OsString` or `std::boxed::Box`.

4.2.6 RVMM-CR-23-105: GuestMemory::try_access() Invariant not Checked

Affected Component: vmm-memory/src/guest_memory.rs

4.2.6.1 Description

X41 found that the trait `GuestMemory` defined in the vm-memory crate exposes ***try_access()***, which takes a callback function as parameter. The method expects the callback to adhere to the invariant to return a size whose value is between 0 and the also provided `len` parameter. However, the callback function may violate this and return lengths higher than the provided length, which is not checked for. Listing 4.5 shows the code responsible for handling the return value of the callback. The variable `cur` stores the current address, which after adding an arbitrary length exceeding `count`, may still be valid and thus result in a hole in the guest memory map.

```
match f(total, len as usize, start, region) {
    Ok(0) => return Ok(total),
    Ok(len) => {
        total += len;
        // `total` may exceed `count` if the callback breaks the invariant
        // The following condition is false, when total > count
        if total == count {
            break;
        }
        cur = match cur.overflowing_add(len as GuestUsize) {
            (GuestAddress(0), _) => GuestAddress(0),
            (result, false) => result,
            (_, true) => panic!("guest address overflow"),
        }
        // We my still end up with a valid `cur` address, but may have
        // skipped a hole.
    }
    e => return e,
}
```

Listing 4.5: Handling Callback Return Value

4.2.6.2 Solution Advice

X41 recommends to produce a runtime error or panic when the provided callback does not observe the invariant to let the implementers know of a bug.

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

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

Acronyms

CWE Common Weakness Enumeration 13