



## Penetration Test Report

Trifecta Tech Foundation

V 1.0

Amsterdam, September 16th, 2025

Confidential

## Document Properties

Client	Trifecta Tech Foundation
Title	Penetration Test Report
Target	<ul style="list-style-type: none"><li>Sudo-rs. A memory safe implementation of sudo and su.</li></ul>
Version	1.0
Pentester	Andrea Jegher
Authors	Andrea Jegher, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

## Version control

Version	Date	Author	Description
0.1	August 21st, 2025	Andrea Jegher	Initial draft
0.2	September 5th, 2025	Marcus Bointon	Review
1.0	September 16th, 2025	Marcus Bointon	1.0

## Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Radically Open Security B.V. is registered at the trade register of the Dutch chamber of commerce under number 60628081.

# Table of Contents

<b>1</b>	<b>Executive Summary</b>	<b>4</b>
1.1	Introduction	4
1.2	Scope of Work	4
1.3	Project Objectives	4
1.4	Timeline	4
1.5	Results In A Nutshell	4
1.6	Summary of Findings	5
1.6.1	Findings by Threat Level	5
1.6.2	Findings by Type	6
1.7	Summary of Recommendations	6
<b>2</b>	<b>Methodology</b>	<b>7</b>
2.1	Planning	7
2.2	Risk Classification	7
<b>3</b>	<b>Reconnaissance and Fingerprinting</b>	<b>9</b>
<b>4</b>	<b>Findings</b>	<b>10</b>
4.1	CLN-008 — "Not yet implemented" panic	10
<b>5</b>	<b>Non-Findings</b>	<b>13</b>
5.1	NF-001 — Sudoers parsing fuzz testing	13
5.2	NF-002 — Side effects in SUID binaries before user authentication	15
5.3	NF-003 — Dynamic test setup	16
5.4	NF-007 — Fuzz testing results	17
<b>6</b>	<b>Future Work</b>	<b>19</b>
<b>7</b>	<b>Conclusion</b>	<b>20</b>
<b>Appendix 1</b>	<b>Testing Team</b>	<b>21</b>

# 1 Executive Summary

## 1.1 Introduction

Between August 12, 2025 and August 25, 2025, Radically Open Security B.V. carried out a penetration test for Trifecta Tech Foundation.

This report contains our findings as well as detailed explanations of exactly how ROS performed the penetration test.

## 1.2 Scope of Work

The scope of the penetration test was limited to the following target:

- Sudo-rs. A memory safe implementation of sudo and su.

The scoped services are broken down as follows:

- Sudo-rs security audit: 6 days
- Report writing: 1 days
- **Total effort: 7 days**

## 1.3 Project Objectives

ROS will perform a code audit of sudo-rs version 0.2.8 with Trifecta Tech Foundation in order to assess the security of its implementation. To do so ROS will access the public repository on [GitHub](#) and guide Trifecta Tech Foundation in attempting to find vulnerabilities, exploiting any such found to try and gain further access and elevated privileges.

## 1.4 Timeline

The security audit took place between August 12, 2025 and August 25, 2025.

## 1.5 Results In A Nutshell

During this crystal-box penetration test we found 1 Low-severity issue.

The audit involved a manual code review of the `sudo-rs` and `su` binaries to identify any potential side effects before user authentication via PAM. This analysis focused on functions related to file system operations, process execution,

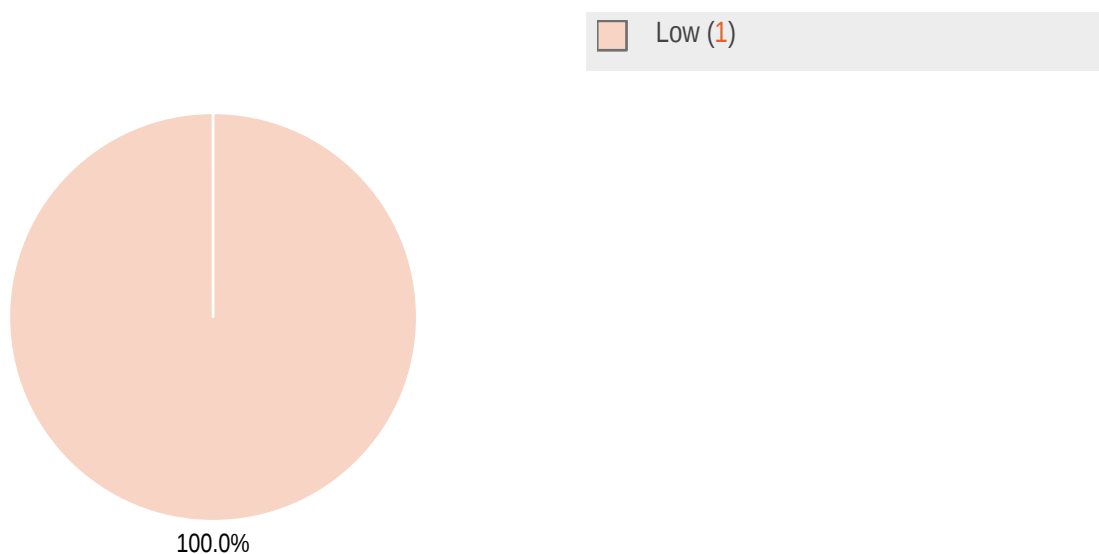
system calls, and environment interactions, using a regex to highlight risky operations in a SUID context. We did not uncover any vulnerabilities.

Additionally, we developed a fuzzing setup to test the parsing of `sudoers` files utilizing AFL (American Fuzzy Lop). This approach fuzzed `sudo-rs` logic for `sudoers` file parsing and policy validation. This approach did not find any vulnerabilities, and resulted only in the creation of a reproducible fuzz testing setup.

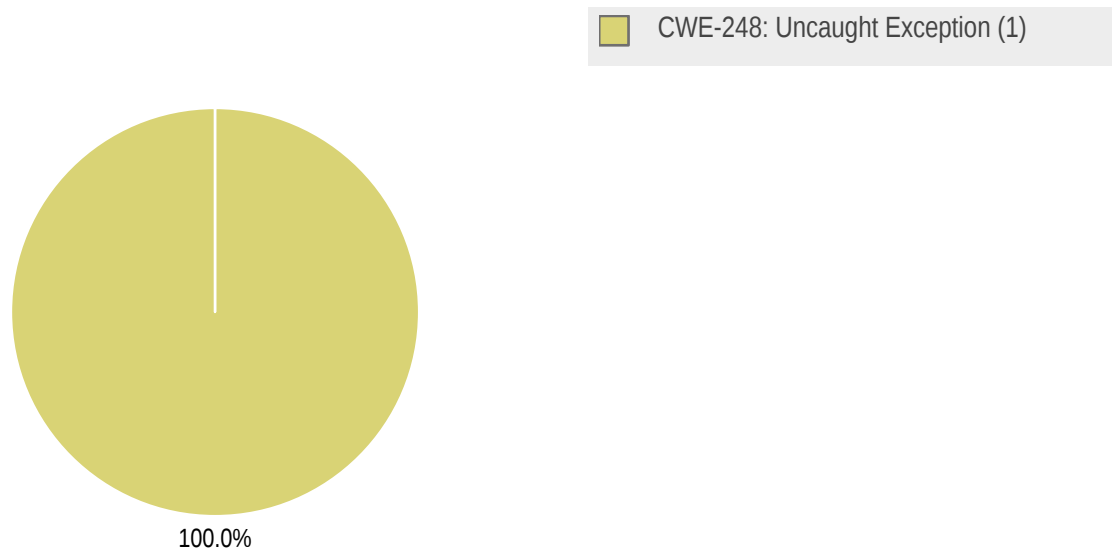
## 1.6 Summary of Findings

Info	Description
<b>CLN-008</b> <b>Low</b> <b>Type:</b> CWE-248: Uncaught Exception <b>Status:</b> none	sudo-rs panics if a sudoers file contains an unimplemented feature.

### 1.6.1 Findings by Threat Level



## 1.6.2 Findings by Type



## 1.7 Summary of Recommendations

Info	Recommendation
<b>CLN-008</b> <b>Low</b> <b>Type:</b> CWE-248: Uncaught Exception <b>Status:</b> none	<ul style="list-style-type: none"><li>Ignore unimplemented features instead of panicking.</li></ul>

## 2 Methodology

### 2.1 Planning

Our general approach during penetration tests is as follows:

1. **Reconnaissance**

We attempt to gather as much information as possible about the target. Reconnaissance can take two forms: active and passive. A passive attack is always the best starting point as this would normally defeat intrusion detection systems and other forms of protection afforded to the app or network. This usually involves trying to discover publicly available information by visiting websites, newsgroups, etc. An active form would be more intrusive, could possibly show up in audit logs and might take the form of a social engineering type of attack.

2. **Enumeration**

We use various fingerprinting tools to determine what hosts are visible on the target network and, more importantly, try to ascertain what services and operating systems they are running. Visible services are researched further to tailor subsequent tests to match.

3. **Scanning**

Vulnerability scanners are used to scan all discovered hosts for known vulnerabilities or weaknesses. The results are analyzed to determine if there are any vulnerabilities that could be exploited to gain access or enhance privileges to target hosts.

4. **Obtaining Access**

We use the results of the scans to assist in attempting to obtain access to target systems and services, or to escalate privileges where access has been obtained (either legitimately through provided credentials, or via vulnerabilities). This may be done surreptitiously (for example to try to evade intrusion detection systems or rate limits) or by more aggressive brute-force methods. This step also consists of manually testing the application against the latest (2021) list of OWASP Top 10 risks. The discovered vulnerabilities from scanning and manual testing are moreover used to further elevate access on the application.

### 2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see: <http://www.pentest-standard.org/index.php/Reporting>

These categories are:

- **Extreme**

Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.

- **High**  
High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**  
Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**  
Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.
- **Low**  
Low risk of security controls being compromised with measurable negative impacts as a result.

### 3 Reconnaissance and Fingerprinting

We were able to gain information about the software and infrastructure through the following automated scans. Any relevant scan output will be referred to in the findings.

- Semgrep – <https://github.com/semgrep/semgrep>
- AFLPlusPlus – <https://aflplusplus/docs/>

## 4 Findings

We have identified the following issues:

### 4.1 CLN-008 — "Not yet implemented" panic

**Vulnerability ID:** CLN-008

**Vulnerability type:** CWE-248: Uncaught Exception

**Threat level:** Low

#### Description:

`sudo-rs` panics if a `sudoers` file contains an unimplemented feature.

#### Technical description:

To reproduce this issue we recommend using a setup similar to what we used in [non-finding NF-003](#) (page 16). Alternatively, it should be enough to edit the `sudoers` file, or another file included by it, adding any of the following examples and run `sudo-rs -l`.

#### File

```
ir}d~r -, #22L) Nrs.d

!#2) NWD
rs.d^\ ^
\
22,WD
WD:, #22.d

%:22,2Lled P = /#u->- 0P = /#u->- 0/bi
```

#### Output

```
/root/additional:3:4: expected host name
!#2) NWD
  ^
/root/additional:4:5: expected host name
rs.d^\ ^
  ^
/root/additional:6:6: expected host name
22,WD
  ^
/root/additional:7:3: expected host name
WD:, #22.d
  ^
/root/additional:9:27: garbage at end of line
```

```
%:22,2lled P = /#u->- 0P = /#u->- 0/bi
      ^
```

```
thread 'main' panicked at src/sudoers/mod.rs:590:14:
not yet implemented
```

## File

```
ireddir , -Or , -i11111122.d
```

```
%:11011111111, !8, !#rrrn8, !ediQ -- = /u-redir -/-2-sdz
"==lundir"-- = /u- = /us- : usr/bi
```

## Output

```
/root/additional:1:27: expected host name
```

```
ireddir , -Or , -i11111122.d
```

```
      ^
/root/additional:4:20: garbage at end of line
```

```
"==lundir"-- = /u- = /us- : usr/bi
      ^
```

```
thread 'main' panicked at src/sudoers/mod.rs:590:14:
not yet implemented
```

## File

```
%:dir -, - = /usr/bi22.d
```

```
%:22, :lledir %
```

## Output

```
/root/additional:3:6: expected user
```

```
%:22, :lledir %
      ^
```

```
thread 'main' panicked at src/sudoers/mod.rs:590:14:
not yet implemented
```

## Trace

For all of these, `sudo-rs` will panic at file `src/sudoers/mod.rs` line 590 with a trace similar to the following:

```
thread 'main' panicked at src/sudoers/mod.rs:590:14:
not yet implemented
stack backtrace:
 0: rust_begin_unwind
 1: core::panicking::panic_fmt
 2: core::panicking::panic
 3: sudo_rs::sudoers::find_item
 4: <core::iter::adapters::flatten::Flatten<I> as core::iter::traits::iterator::Iterator>::next
 5: sudo_rs::sudo::sudo_process
 6: sudo::main
```

note: Some details are omitted, run with `RUST\_BACKTRACE=full` for a verbose backtrace.

## Impact:

If `/etc/sudoers` contains a policy that contains an unimplemented feature, `sudo-rs` will panic while evaluating it.

## Recommendation:

- Ignore unimplemented features instead of panicking.

## 5 Non-Findings

In this section we list some of the things that were tried but turned out to be dead ends.

### 5.1 NF-001 — Sudoers parsing fuzz testing

#### Fuzzing of `sudo -l`

This setup tries to find any `sudoers` file that will allow a user to run a command with `sudo` or that makes `sudo -rs` panic. Note, among these results there will be legitimate `sudoers` files, but we might be able to find parsing errors too, if they exist.

This approach uses a NixOS VM to create a reproducible environment for fuzzing the `sudo-rs` binary. The VM installs Rust, `cargo-afl`, and other build tools declaratively, ensuring consistent builds. Root wrappers are configured so `sudo-rs` behaves like the real `sudo`, while a PAM (Pluggable Authentication Modules) rule allows the `test` user to authenticate without a password to prevent the fuzzer from hanging, simulating a legitimate user entering their password.

Inside the VM, the fuzzer is built with AFL instrumentation and run against the `-l` option of `sudo-rs`, the list command, to perform `sudoers` file parsing and policy validation. Only one instance runs at a time to avoid concurrent writes to the test `sudoers` file, and avoiding false positives, since even with different rules files they are all included at the same time. Additionally, running in a VM isolates the host system from any privileged side effects that might occur.

The setup is flexible and extensible. In the future it could fuzz environment variables or other command line options through different targets. Overall, it provides a controlled, repeatable way to test root-level logic safely while preserving reproducibility through NixOS configuration.

For more details see [the public repository](#).

Below we report the fuzzer target that writes the fuzzer data to `/root/additional` and then calls `sudo_rs::sudo_main`. Note that we called this program with the `-l` argument. In this way when `sudo_rs::sudo_main` calls for `std::env::args()` will see the list action.

```
#[macro_use]
extern crate afl;
extern crate sudo_rs;

use std::{fs::File, io::Write, path::PathBuf};
const BASE_PATH: &'static str = "/var/run/sudo-rs/ts";

fn main() {
    fuzz!(|data: &[u8]| {
        let path = "/root/additional";
        if let Ok(mut output) = File::create(path) {
            if let Ok(s) = std::str::from_utf8(data) {
                match write!(output, "{}", s) {
                    Ok(_) => {
                        sudo_rs::sudo_main();
                        unsafe {
                            let uid = libc::getuid();
                        }
                    }
                }
            }
        }
    })
}
```

```
// a successful login will create a session file
// that might create false positives
let mut path = PathBuf::from(BASE_PATH);
path.push(uid.to_string());
let _ = std::fs::remove_file(path);

panic!("User passed after sudo -l {}", uid);
}
}
Err(_) => {
    return;
}
}
}
});
}
```

The `sudoers` file is shown below, so it will include the `/root/additional` file. As you see there is no entry for user `test` so `sudo-rs -l` will exit saying that `test` cannot run any command, and it will call `std::process::exit(1)`; at line 153 of `sudo-rs/src/sudo/mod.rs`. Instead, if `sudo -l` was successful, and there was a rule allowing `test` to run `sudo`, it will hit the panic triggering a crash in the fuzzer. Of course this setup could be improved, and more checks could be performed after `test` passed the `sudo-rs -l` command before triggering the panic.

```
#includedir /root/  
root      ALL=(ALL:ALL) ALL  
%wheel    ALL=(ALL:ALL) ALL
```

Another important piece of the configuration is a PAM policy for `sudo-rs`. This always grants access to user `test` when the `sudo` binary asks for authentication. In this way the fuzzer will not hang asking for a password when a valid `sudoers` configuration for `test` is found that would require the `test` user to authenticate itself with its own password.

```
auth sufficient pam_succeed_if.so use_uid user = test
```

## Fuzzing visudo file check

The `sudoers` parsing library is shared between both `sudo` and `visudo`. Because of this, `visudo` can be used as a fuzzing target to exercise the parsing logic of `sudoers` files. For example, running `visudo -c -f path/to/sudoers` will check the validity of a given `sudoers` file.

```
visudo -c -f path/to/sudoers
```

## Setup

Build the project with AFL and run `visudo` under the fuzzer with a sample input directory and an output directory for findings.

```
cargo afl build
cargo afl fuzz -in ./in -o ./out ./target/debug/visudo -c -f @@
```

Where in the `in` folder are several sample `sudoers` configuration files.

## Goals and limitations

This approach mainly will check whether the parser safely consumes input. It verifies whether `sudo-rs` interprets the resulting policies correctly, but only in a limited way. A file that is syntactically valid but semantically misleading could go undetected if the target user is not `test`.

## 5.2 NF-002 — Side effects in SUID binaries before user authentication

Potentially dangerous side effects in SUID binaries may occur before authenticating through PAM (Pluggable Authentication Modules). For example, if a SUID binary writes a file whose path is controlled by the user calling it.

We focused on *sink* functions or system calls that can have side effects, such as modifying the filesystem, changing permissions, executing processes, or interacting with system resources. We compiled a regex to use during manual code analysis to find the following:

1. Filesystem operations: `File::create`, `File::open`, `fs::create_dir`, `remove_file`, `remove_dir_all`, `truncate`, `set_permissions`
2. Process operations: `fork`, `exec`, `Command::new`, `kill`, `setuid`, `setgid`, `prctl`
3. System calls & libc wrappers: `libc::openat`, `libc::fchown`, `libc::chmod`, `libc::syscall`, `ioctl`
4. Environment and configuration access: `env::`, `passwd()`, `getpwuid_r()`, `getpwnam_r()`, `getgrnam_r()`, `sysctl()`
5. I/O operations: `read`, `write_all`, `.write()`, `recv`, `send`, `FileLock::exclusive()`
6. Device and terminal operations: `tcgetattr`, `tcsetpgrp`, `openpty`, `isatty`, `ttynam_r`

During the analysis we identified places in the binary where these 'sink' operations are executed before authentication, making sure that it was not possible to use them in a malicious way.

Here is the full regex:

```
[^\\w)((uid\\(|gid\\(|DirBuilder\\(|chown\\(|Command::new\\(|create_dir\\(|exec\\(|fchown\\(|File::create\\(|FileLock::exclusive\\(|File::new\\(|File::open\\(|File::open_for_user\\(|File::options\\(|fork\\(|fs::canonicalize\\(|fs::create_dir\\(|fset\\(|fsetispeed\\(|fsetospeed\\(|fs::File::create\\(|fs::File::open\\(|fs::File::options\\(|fs::metadata\\(|fs::OpenOptions::new\\(|fs::read\\(|fs::read_dir\\(|fs::read_to_string\\(|fs::remove_dir_all\\(|fs::remove_file\\(|fstat\\(|ioctl\\(|kill\\(|libc::abort\\(|libc::calloc\\(|libc::chown\\(|libc::clock_gettime\\(|libc::close_range\\(|libc::c_uint::from\\(|libc::dev_t::from_le_bytes\\(|libc::dlopen\\(|libc::dlsym\\(|libc::_exit\\(|libc::fchown\\(|libc::fcntl\\(|libc::flock\\(|libc::fork\\(|libc::free\\(|libc::fstat\\(|libc::getgid\\(|libc::geteuid\\(|libc::getgid\\(|libc::getgrgid_r\\(|libc::getgrnam_r\\(|libc::getgrouplist\\(|libc::getgroups\\(|libc::gethostname\\(|libc::getpgid\\(|libc::getpgrp\\(|libc::getpwnam_r\\(|libc::getpwuid_r\\(|libc::getsid\\(|libc::getuid\\(|libc::ioctl\\(|libc::isatty\\(|libc::kill\\(|libc::killpg\\(|libc::mkdtemp\\(|libc::openat\\(|libc::openpty\\(|libc::pid_t::from_le_bytes\\(|libc::poll\\(|libc::read\\(|libc::recv\\(|libc::send\\(|libc::setgid\\(|libc::setgroups\\(|libc::setpgid\\(|libc::setresgid\\(|libc::setresuid\\(|libc::setuid\\(|libc::setuid\\(|libc::sigaction\\(|libc::sigemptyset\\(|libc::sigfillset\\(|libc::sigprocmask\\(|libc::syscall\\(|libc::sysconf\\(|libc::sysctl\\(|libc::syslog\\(|libc::tcgetattr\\(|libc::tcsetpgrp\\(|libc::tcgetsid\\(|libc::tcsetpgrp\\(|libc::ttynam_r\\(|libc::uid_t::from_le_bytes\\(|libc::umask\\(|libc::waitpid\\(|open\\(|OpenOptions::\\(|passwd\\(|prctl\\(|remove_dir_all\\(|remove_file\\(|
```

```
(setgid\()|(set_len\()|(set_permissions\()|(setsid\()|(setuid\()|(shutdown\()|(sysctl\()|(truncate\()|(\.write\()|(write_all\()|(env::(\w|::)+\()))
```

## 5.3 NF-003 — Dynamic test setup

We used a VM build with NixOS to create a reproducible test environment. Encapsulating tests within a VM isolates the system from potential side effects that could affect the host machine and allows restarting from a clean setup every time. For example, we could easily test configuration with multiple users without having to create them on the host machine, or we could easily add policies in a folder included in `/etc/sudoers`.

We include the `/root` directory in the `sudoers` file so that it's easy to any test configuration at runtime, especially since file `/etc/sudoers` is read-only in NixOS.

This is the Nix configuration, or `configuration.nix`, that contains a declarative system configuration:

```
{ config, pkgs, lib, ... }: {
  nix.nixPath = [ "nixpkgs=${builtins.storePath <nixpkgs>}" ];

  environment = {
    systemPackages = with pkgs; [ rush tmux sudo-rs vim openssl ];
  };

  security.sudo.enable = false;
  security.wrappers.sudo = {
    source = "${pkgs.sudo-rs}/bin/sudo";
    owner = "root";
    group = "root";
    permissions = "u+rs,g+x,o+x";
  };
  security.wrappers.su = lib.mkForce {
    source = "${pkgs.sudo-rs}/bin/su";
    owner = "root";
    group = "root";
    permissions = "u+rs,g+x,o+x";
  };

  security.pam.services.sudo-rs.text = ''
    auth      include login
    account   include login
    password  include login
    session   include login
  '';

  environment.etc."nixos/configuration.nix".source = ./configuration.nix;
  environment.etc."pam.d/sudo".source = "/etc/pam.d/sudo-rs";
  environment.etc."sudoers".text = ''
    #includedir /root/
    root      ALL=(ALL:ALL) ALL
    %wheel    ALL=(ALL:ALL) ALL
  '';

  system.stateVersion = "25.05";

  users = {
    users."test" = {
      home = "/home/test";
    };
  };
}
```

```

    password = "test";
    isNormalUser = true;
  };
  users.root = { password = "root"; };

};

services.getty.autologinUser = "test";
services.getty.autologinOnce = true;
}

```

To create a VM from this file and start it, use the following script. Note that **Nix must be installed** beforehand.

```

#!/usr/bin/env bash

set -ex

# Nix-build will create a disk too
# you may remove it to have a clean start, needed some times
# rm -f ./nixos.qcow2
nix-build '<nixpkgs/nixos>' -A vm -I nixos-config=./configuration.nix

# If you already build it you can just run this to start it
./result/bin/run-nixos-vm \
  -display none \
  -serial mon:stdio \
  -enable-kvm \
  -cpu host \
  -m 4G

```

## 5.4 NF-007 — Fuzz testing results

Fuzz testing found two developer-controlled panic locations, where the developers intentionally raised a panic, and one type of hang (inputs for which the target runs for more than the maximum allowed time).

### Not implemented panic

Line `/home/test/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/sudo-rs-0.2.8/src/sudoers/mod.rs:590:14` raises a panic with reason "not implemented yet".

An example is the following, but this file, with only one line, looks more like random characters than a real policy. Additionally, this panic is not uncontrolled and was written by the developers.

```
%:_Alias PKGWWWD=MI,St:
```

### Forced panic for a rule valid for user `test`

Line `src/bin/fuzz_sudo.rs:27:21` raises the forced panic in the fuzzer target, meaning that the fuzzer created a configuration for which user `test` can run `sudo -l`, causing the panic. Note, that these will not make `sudo-rs` panic, as one goal of the target is to find valid policies too.

Many of these were due to a user alias assigning all users to the ADMINS alias, which is not interesting, and led us to create a second target that excluded this case.

Excluding them allowed us to find other valid policies for user `test` similar to the following example, featuring complicated but valid rules. However, the fuzzer did not find rules with effects other than the one intended, and they all worked because of the ALL keyword, which might be worth excluding in new targets.

```
#37,#37,ALL, !D, ALL, !DLL, ALL, !D, ALLLLdeploy ALL=(ALL) ALL, !/usdi, !iD
```

## Hangs

Lastly, the fuzzer found a few hangs too, all related to lines similar to `#includedir /etc`, so to the inclusion of folder `/etc`. But in a real scenario `sudo-rs` simply logs a list of errors. These are identified as hangs because the fuzzer has an execution timeout during which `sudo-rs` cannot read all files in `/etc`.

An example is:

```
#includedir /etc
User_Alibs ADMINS =LL, ALL, !D
```

## 6 Future Work

- **Retest of findings**

When mitigations for the vulnerabilities described in this report have been deployed, perform a repeat test to ensure that they are effective and have not introduced other security problems.

- **Regular security assessments**

Security is a process that must be continuously evaluated and improved; this penetration test is just a single snapshot. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security.

## 7 Conclusion

The audit, encompassing both static analysis and dynamic fuzz testing, did not uncover any security vulnerabilities.

The establishment of reproducible test environments, including a NixOS VM setup and AFL-based fuzzing configurations, significantly contributed to the transparency and rigor of this audit. These setups provide a foundation for future testing and continuous improvement, ensuring that any future issues can be detected early.

In the non-findings section we describe reproducible test and fuzzing setups that we [published](#).

We recommend fixing all of the issues found and then performing a retest in order to ensure that mitigations are effective and that no new vulnerabilities have been introduced.

Finally, we want to emphasize that security is a process that must be continuously evaluated and improved – this penetration test is just a one-time snapshot. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security. We hope that this pentest report (and the detailed explanations of our findings) will contribute meaningfully towards that end.

Please don't hesitate to let us know if you have any further questions, or need further clarification on anything in this report.

## Appendix 1 Testing Team

Andrea Jegher	Andrea is a security engineer with experience in offensive security and secure development. He started his career focusing on web application as a developer and as a penetration tester. Later he studied other fields of security such as cloud, networks, and desktop applications.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.

Front page image by Slava (<https://secure.flickr.com/photos/slava/496607907/>), "Mango HaX0ring",  
Image styling by Patricia Piolon, <https://creativecommons.org/licenses/by-sa/2.0/legalcode>.