RADICALLY OPEN SECURITY

Secure Code Review Report

NLnet – Tweede Golf

V 1.0
Amsterdam, September 25th, 2023
Public

Document Properties

Client	NLnet – Tweede Golf
Title	Secure Code Review Report
Target	sudo-rs main branch (last commit b5eb2c654f8971a7191d341228f06d58ba3746ff)
Version	1.0
Pentester	Andrea Jegher
Authors	Andrea Jegher, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	

Version control

Version	Date	Author	Description
0.1	September 15th, 2023	Andrea Jegher	Initial draft
0.2	September 20th, 2023	Marcus Bointon	Review
1.0	September 25th, 2023	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

Table of Contents

1 Executive Summary

1.1 Introduction

Between September 4, 2023 and September 15, 2023, Radically Open Security B.V. carried out a code review for NLnet – Tweede Golf.

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

1.2 Scope of work

The scope of the code review was limited to the following target:

- sudo-rs main branch (last commit b5eb2c654f8971a7191d341228f06d58ba3746ff)

The scoped services are broken down as follows:

- Code Review: 9 days
- Report Writing: 1 days
- **Total effort: 10 days**

1.3 Project objectives

ROS will perform a code review of `sudo-rs` with Tweede Golf in order to assess the security of the implementation. To do so ROS will access the public repository on GitHub, and guide Tweede Golf in attempting to find vulnerabilities in the code, exploiting any such found to try and gain further access and elevated privileges.

1.4 Timeline

The code review took place between September 4, 2023 and September 15, 2023.

1.5 Results In A Nutshell

During this code review we found 1 Moderate and 2 Low-severity issues.

The analysis covered the `main` branch of the GitHub repository for sudo-rs, and mostly focused on verifying that it was not possible to perform privileged actions without proper authorization. Additionally, the tester used automatic fuzzing tools on specific functions that receive direct user input, trying to identify potential crashes.

The audit consisted of manual code review, following the execution flow, looking for possible user inputs that could alter the flow and bypass security checks. The program uses libc correctly to handle PAM authentication, which is one of the most critical parts where user input could affect the execution flow. Another focus of the test was to review the uses of `unsafe` rust code; the tester did not find any opportunities for misuse or other issues, and it was mostly used only to wrap calls to libc, making the attack surface very small.

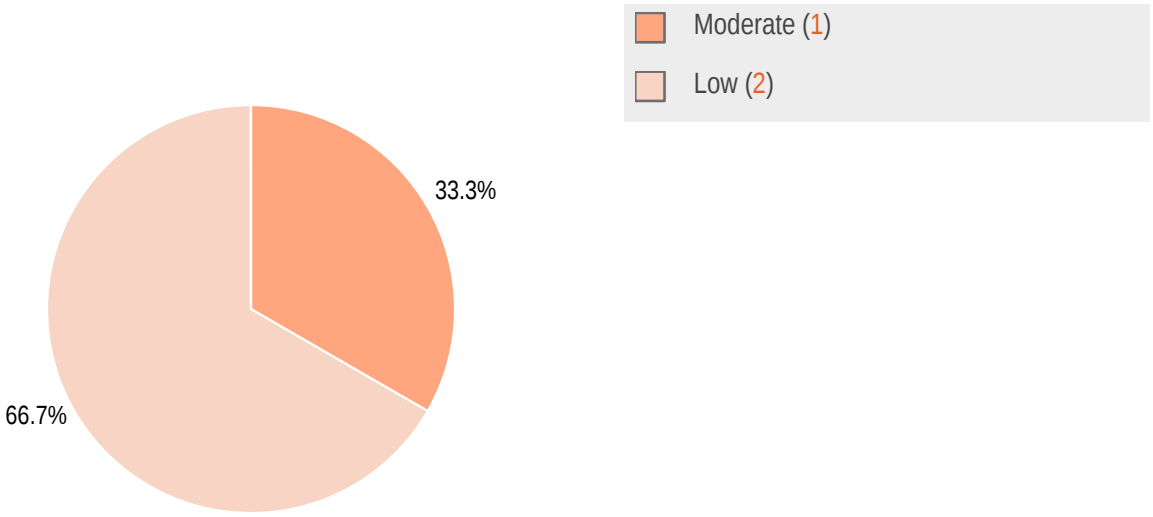
We found a path traversal vulnerability in CLN-001 (page 10) where a user that can control their username could corrupt arbitrary files on the file system. This happens because the `sudo-rs` program creates a path for a timestamp file from the username of the user requesting privileged access through `sudo-rs`, and the username is not sanitized or escaped properly when doing so.

We reported two other configuration issues in CLN-003 (page 12) and CLN-004 (page 13) that do not pose a direct threat to `sudo-rs` but could impact the security of the program in some corner cases.

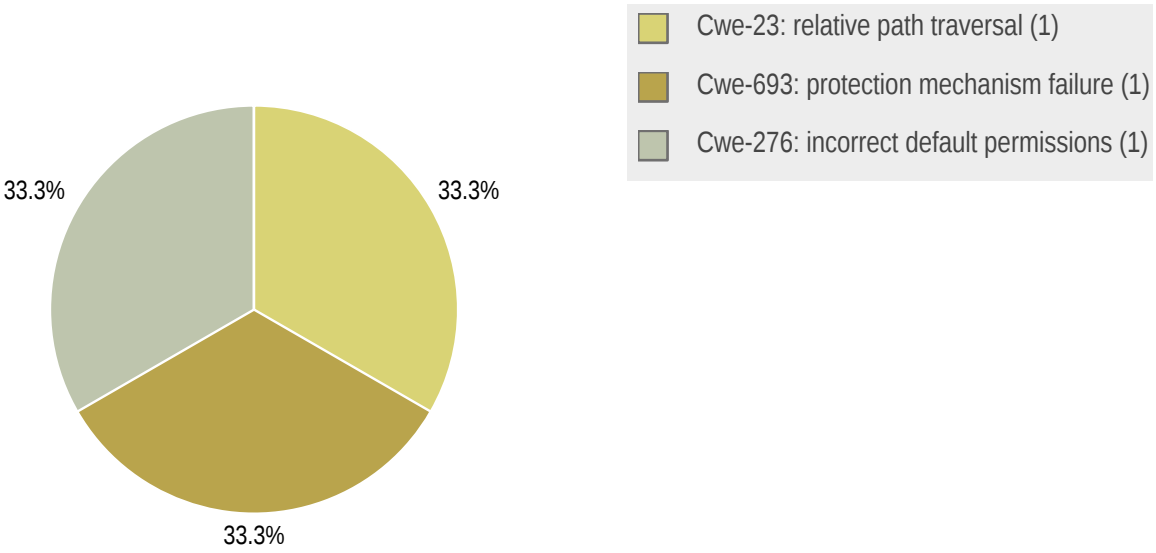
1.6 Summary of Findings

ID	Type	Description	Threat level
CLN-001	CWE-23: Relative Path Traversal	Usernames containing special characters are allowed, enabling path traversal through usernames.	Moderate
CLN-003	CWE-693: Protection Mechanism Failure	The cargo release build does not strip symbols, so they will be included in the final binary.	Low
CLN-004	CWE-276: Incorrect Default Permissions	A function used to invoke the libc chown function uses a default user ID value instead of failing.	Low

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

ID	Type	Recommendation
CLN-001	CWE-23: Relative Path Traversal	• Escape `/` and `.` characters in usernames before concatenating them to the `BASE_PATH`. • Append a slash (`/`) character to the `BASE_PATH` constant. • Canonicalize the resulting path. • Check that the path obtained after concatenation and canonicalization starts with the `BASE_PATH` constant. • Alternatively, use the user's `uid`, which does not have this opportunity for abuse.
CLN-003	CWE-693: Protection Mechanism Failure	• Change the release build configuration to strip symbols.
CLN-004	CWE-276: Incorrect Default Permissions	• If the `unwrap` call does not succeed, the method should fail, or the default value should depend on the system configuration to make sure it is not possible to use.

2 Methodology

2.1 Planning

During the code audit, we take the following approach:

1. **Thorough comprehension of functionality**
   We try to get a thorough comprehension of how the application works and how it interacts with the user and other systems. Having detailed documentation (manuals, flow charts, system sequence diagrams, design documentation) at this stage is very helpful, as they aid the understanding of the application
2. **Comprehensive code reading**
   goals of the comprehensive code reading are:
   - to get an understanding of the whole code
   - identify adversary controlled inputs and trace their paths
   - identify issues

3. **Static analysis**
   Using the understanding we gained in the previous step, we will use static code analysis to uncover any vulnerabilities. Static analysis means the specialist will analyze the code and implementation of security controls to get an understanding of the security of the application, rather than running the application to reach the same goal. This is primarily a manual process, where the specialist relies on his knowledge and expertise to find the flaws in the application. The specialist may be aided in this process by automatic analysis tools, but his or her skills are the driving force.
   Depending on the type of application, we will identify the endpoints. In this case, it means where data enters and leaves the application. The data is then followed through the application and is leading in determining if assessing the quality of the security measures.
4. **Dynamic analysis**
   Dynamic analysis can also be performed. In this case, the program is run and actively exploited by the specialist. This is usually done to confirm a vulnerability and as such follows the result of the static analysis.
5. **Fuzzing**
   Fuzz testing or Fuzzing is a software testing technique which in essence consists of finding implementation bugs using malformed/semi-malformed data injection in an automated fashion.
6. **Concolic analysis**
   If the specialist thinks it useful, additional concolic analysis may be performed on selected subsets of the code.

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 Findings

We have identified the following issues:

3.1 CLN-001 — Session file path traversal

Vulnerability ID: CLN-001

Vulnerability type: CWE-23: Relative Path Traversal

Threat level: Moderate

Description:

Usernames containing special characters are allowed, enabling path traversal through usernames.

Technical description:

The program creates the timestamp file path by concatenating the username with a default base path without verifying that the base path is the default one after the concatenation. If a username contains slash (`/`) and dot (`.`) characters, it is possible to corrupt arbitrary files.

The `sudo-rs` command implements actions `-K` and `-k` that will delete or reset the timestamp file used to keep track of the user `sudo-rs` command invocations. This file is located at `/var/run/sudo-rs/ts` and is owned by root.

The `sudo-rs` handles these actions in the `open_for_user` method on lines 92 to 104 of `src/sudo/mod.rs` to obtain the session file corresponding to the user that is invoking `sudo-rs`. These functions *do not require authentication*, as they are only meant to edit the timestamp file owned by the user invoking the command. They also don't need to be in the `sudo` group, or to have a valid entry in the `sudoers` file to access these actions, but it provides the same behavior as the standard `sudo` command.

```
SudoAction::RemoveTimestamp => {
    let user = resolve_current_user()?;
    let mut record_file =
        SessionRecordFile::open_for_user(&user.name, Duration::seconds(0))?;
    record_file.reset()?;
    Ok(())
}
SudoAction::ResetTimestamp => {
    if let Some(scope) = RecordScope::for_process(&Process::new()) {
        let user = resolve_current_user()?;
        let mut record_file =
            SessionRecordFile::open_for_user(&user.name, Duration::seconds(0))?;
        record_file.disable(scope, None)?;
    }
    Ok(())
```

```
}
```

The method `open_for_user`, defined from line 47 of `src/system/timestamp.rs`, is shown here:

```
const BASE_PATH: &'static str = "/var/run/sudo-rs/ts";

pub fn open_for_user(user: &'u str, timeout: Duration) -> io::Result<Self> {
    let mut path = PathBuf::from(Self::BASE_PATH);
    path.push(user);
    SessionRecordFile::new(user, secure_open_cookie_file(&path)?, timeout)
}
```

In this snippet, the line `path.push(user);` concatenates the `user` argument (a string containing the username of the user invoking `sudo-rs`) with the constant `BASE_PATH` containing the timestamp files root folder. There is no additional check on the username or on the result of the `push` operation.

It is possible for a username on a Linux-based OS to contain slash (`/`) and dot (`.`) characters which have special meanings in path names. The `useradd` command does not allow these characters by default, but that can be overridden using the `badname` option, so it is possible to create one with the following command `sudo useradd '../path/traversal' --badname`.

For example, an attacker could create a user with username `../../../../bin/cp` and invoke `sudo-rs -K` to reset its own timestamp file, but since the username contains a path traversal it is file `/bin/cp` that will be overwritten with an empty timestamp file.

Similarly, it could be possible to reset other users timestamp files. For example, if a user `victim` has a valid session, an attacker with username `../ts/victim` could reset their timestamp by invoking `sudo-rs -K`.

Impact:

An attacker that can create users with names that contain slash (`/`) and dot (`.`) characters, could corrupt arbitrary files by using the `sudo-rs -K` command.

However, an attacker would need to be able to create users with arbitrary names on the OS, which usually requires root privileges.

Recommendation:

The `open_for_user` method should:

1. Escape `/` and `.` characters in usernames before concatenating them to the `BASE_PATH`.
2. Append a slash (`/`) character to the `BASE_PATH` constant.
3. Use the canonicalize function to resolve all intermediate components in paths and symbolic links into an absolute path without traversal.
4. Check that the path obtained after concatenation and canonicalization starts with the `BASE_PATH` constant.

Alternatively, do not use the username to create the timestamp file path but instead use the user's `uid`, which does not have this opportunity for abuse.

3.2 CLN-003 — Release build does not strip symbols

Vulnerability ID: CLN-003

Vulnerability type: CWE-693: Protection Mechanism Failure

Threat level: Low

Description:

The cargo release build does not strip symbols, so they will be included in the final binary.

Technical description:

The `cargo` release build configuration is shown below:

```
[profile.release]
strip = "debuginfo"
lto = true
opt-level = "s"
```

This configuration strips debug information, but not all the symbols. It is possible to check this using the checksec script. That will create an output similar to the following:

```
   ~/Tools/checksec.sh/checksec --file=./target/release/sudo
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols
E
Full RELRO      No canary found   NX enabled    PIE enabled     No RPATH   No RUNPATH   1779 Symbols
```

For comparison, the output of this tool on the original `sudo` binary shows that it does not contain symbols:

```
   ~/Tools/checksec.sh/checksec --file=/usr/bin/sudo
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols
E
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   RW-RUNPATH   No Symbols
```

Impact:

Since the code is open source, there is not much information to be gained, but removing these symbols might make reverse engineering of the binary harder.

Recommendation:

Change the release build configuration to strip symbols:

```
[profile.release]
strip = "symbols"
lto = true
opt-level = "s"
```

3.3 CLN-004 — Chown failure handling

Vulnerability ID: CLN-004

Vulnerability type: CWE-276: Incorrect Default Permissions

Threat level: Low

Description:

A function used to invoke the `libc` `chown` function uses a default user ID value instead of failing.

Technical description:

The following snippet from `src/system/mod.rs` at line 243 implements the `chown` function:

```
pub fn chown<S: AsRef<CStr>>(
    path: &S,
    uid: impl Into<Option<UserId>>,
    gid: impl Into<Option<GroupId>>,
) -> io::Result<()> {
    let path = path.as_ref().as_ptr();
    let uid = uid.into().unwrap_or(UserId::MAX);
    let gid = gid.into().unwrap_or(GroupId::MAX);

    cerr(unsafe { libc::chown(path, uid, gid) }).map(|_| ())
}
```

This function wraps a call to the `libc` function `chown` to set the owner of a file. The code selects the new owner at lines 249 and 250:

```
let uid = uid.into().unwrap_or(UserId::MAX);
let gid = gid.into().unwrap_or(GroupId::MAX);
```

The values will be either the `uid`, and `gid` given in the arguments, or in case of an error during the variable unwrap `UserId::MAX`, and `GroupId::MAX`.

Impact:

A failure in the `unwrap` function might result in unexpected (and wrong) file ownership, and if there is a user with the default user ID, they will have control of these files.

However, in most Unix configurations, it is not possible to create a user with the current `UserId::MAX` ID.

Recommendation:

- If the `unwrap` call does not succeed, the method should fail, or the default value should depend on the system configuration to make sure it is not possible to use.

4 Non-Findings

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

4.1 NF-002 — Fuzzing Visudo Sudoers Parser

Fuzz testing is an automated approach to uncovering defects and potential security weaknesses by supplying arbitrary input to a program or function. Typically, this method is paired with a tool capable of recognizing erroneous or undefined behavior, such as accessing memory beyond its bounds (buffer overflow/underflow), employing uninitialized data, accessing memory after it's been deallocated, or releasing the same memory multiple times.

During the audit, the tester used cargo-fuzz and cargo-afl crates to instrument and perform fuzzing of the sudo-rs sudoers policy parser.

The two frameworks required different setups:

1. Cargo Fuzz

This utility requires creating a fuzzing target that imports public functions from the target Rust crate. This is an issue since the `sudo-rs` crate only exports three functions that are not ideal fuzzing targets. So, the tester modified the code in order to make all the functions public at create level, and implemented a fuzzing target similar to the following:

```
#![no_main]

use libfuzzer_sys::fuzz_target;
extern crate sudo_rs;

fuzz_target!(|data: &[u8]| {
    if let Ok(s) = std::str::from_utf8(data) {
        use sudo_rs::sudoers::basic_parser::parse_lines;
        let _result: Vec<sudo_rs::sudoers::basic_parser::Parsed<sudo_rs::sudoers::Sudo>> =
 parse_lines(&mut sudo_rs::sudoers::char_stream::PeekableWithPos::new(s.chars()));
    }
});
```

```
#![no_main]

use libfuzzer_sys::fuzz_target;
extern crate sudo_rs;

fuzz_target!(|data: &[u8]| {
    if let Ok(s) = std::str::from_utf8(data) {
        let args: Vec<String> = s.split_whitespace().map(|s| s.to_owned()).collect();
        let _result = sudo_rs::cli::SudoOptions::try_parse_from(args);
    }
});
```

2. AFL (American Fuzzy Lop)

AFL take a different approach that requires an instrumented binary that takes a file as input, making the `visudo` program ideal for this kind of testing, especially the `check` feature offered by that program that will parse and validate an input sudoers file.

The tester used the following commands to instrument the binary:

```
# Install the cargo-afl utility
cargo install cargo-afl

# Build an instrumented release
cd sudo-rs
cargo fuzz build --release

cd ..
mkdir in out
sudo cp /etc/sudoers ./in
sudo chown USER:USER ./in/sudiers
afl-fuzz -i in -o out -D -- ./sudo-rs/target/release/visudo -c -f @@
```

We did not find any security issues from running these fuzzers.

5 Future Work

- **Retest of findings**
  When mitigation for the vulnerabilities described in this report have been deployed, a repeat test should be performed to ensure that they are effective and have not introduced other security problems.
- **Regular security assessments**
  Security is an ongoing process and not a product, so we advise undertaking regular security assessments and penetration tests, ideally prior to every major release or every quarter.

6 Conclusion

During this code review we found 1 Moderate and 2 Low-severity issues.

We found the `sudo-rs` code sound and robust. It implements permissions checks correctly, and relies on well-tested external libraries to handle the most delicate processes, such as user authentication. The one moderate-severity issue stems from a lack of input validation in a corner case not easily found in practice, and the developers promptly created a fix for it soon after we reported it. The low-severity issues relate to a minor packaging issue and an ownership edge-case that's unlikely to present a security problem.

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

Finally, we want to emphasize that security is a process – this penetration test is just a one-time snapshot. Security posture must be continuously evaluated and improved. 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 Jegher is an information security professional focused on web application security.