

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Secpoline: A Scalable Approach to Build Secure In-Process Syscall Interposers

Ruben Sturm and Anton Schelfhout, *DistriNet, KU Leuven;* Merve Gülmez, *Ericsson Security Research;* Adriaan Jacobs and Stijn Volckaert, *DistriNet, KU Leuven*

https://www.usenix.org/conference/usenixsecurity26/presentation/sturm

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology





USENIX Security '26 Artifact Appendix: Secpoline: A Scalable Approach to Build Secure In-Process Syscall Interposers

Ruben Sturm
DistriNet, KU Leuven

Anton Schelfhout
DistriNet, KU Leuven

Merve Gülmez
Ericsson Security Research

Adriaan Jacobs
DistriNet, KU Leuven

Stijn Volckaert
DistriNet, KU Leuven

A Artifact Appendix

A.1 Abstract

Secpoline is a high-performance secure in-process syscall interposer. The artifact concists of the source code for Secpoline and a collection of case studies and benchmarks to support our claims. The evaluation consists of four main components:

Security Evaluation: We implement proof-of-concept attacks targeting Secpoline based on a set of known attacks collected from prior work (Section 7.1).

Performance Evaluation: We evaluate the performance overhead of Secpoline compared to the state-of-the-art secure in-process syscall interposer (Section 7.2) and MPK sandbox (Section 7.3). To do so, we adapt the benchmarks used by these existing prototypes.

Falco Case Study: We implement a kernel-module-free version of the Falco intrusion detection engine and evaluate it (Section 8.1).

ProxySQL Case Study: We integrate ProxySQL into Secpoline to implement an in-process sidecar (Section 8.2).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

In general, running the code in this artifact does not pose any security or privacy risks.

The only potential concern arises when running the zip benchmark for the Falco case study, as it installs a kernel module that intercepts system calls system-wide. This is a public and well-tested kernel module provided by Falco (https://falco.org/docs/concepts/event-sources/kernel/), and it is automatically uninstalled at the end of the benchmark.

A.2.2 How to access

The source code required for this artifact is available at https://github.com/lazypoline/secpoline. This repository contains three branches, the main branch contains the source code for the baseline version of Secpoline and a branch for each case study evaluated (ProxySQL and Falco).

A.2.3 Hardware dependencies

The artifact requires an x86_64 CPU that supports Intel MPK.

A.2.4 Software dependencies

This artifact requires a Linux system with kernel version 6.12+ and CMake version 3.16+. The artifact has been tested on Ubuntu 20.04 and Ubuntu 24.04. Part of the artifact requires a modified version of the Linux kernel. This artifact contains a patch for Linux kernel version 6.12.0 to create this modified kernel.

A.2.5 Benchmarks

This artifact requires the Cerberus benchmarks (https://github.com/ku-leuven-msec/The-Cerberus-Project), the endokernel benchmarks (https://github.com/endokernel/test) and the apache httpd server (https://github.com/apache/httpd).

A.3 Set-up

Secpoline requires access to `mmap` at low virtual addresses and the ability to place hardware breakpoints. These features can be enabled using:

```
echo 0 | sudo tee /proc/sys/vm/mmap_min_addr
echo 2 | sudo tee
    /proc/sys/kernel/perf_event_paranoid
```

A.3.1 Installation

More detailed instructions can be found in the README of each branch.

To test the base functionality of Secpoline, use the code in the main branch. Start by installing the required dependencies:

```
apt install libaudit-dev musl-tools clang
    cmake libssl-dev
```

Then build Secpoline using CMake:

```
# From the root of the repository
mkdir build
cd build
cmake ../
make -j
```

Now Secpoline can be run as follows:

```
path/to/output/libloader.so
    path/to/output/secpoline <some binary>
```

A.3.2 Basic Test

A basic test example is provided that is built alongside Secpoline. To verify whether Secpoline is actually running and intercepting syscalls, enable the `POST_PRINT_SYSCALLS` option in `src/include/config.h`.

Now, from the `output` folder, run:

```
./libloader.so ./secpoline ./main
```

Secpoline can be tested on other applications as well, for example:

```
./libloader.so ./secpoline /usr/bin/ls
```

This should print all syscalls invoked by the application, including their arguments and return values. example output: `[tid 3079692] openat(0xffffff9c, 0x7fe399fd48b0, 0x80000, 0x0, 0x7fe39ac66010, 0x7) = 0x5`

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Secpoline achieves parity with the collective security guarantees of existing PKU-protected syscall interposers. This is proven by experiment (E1) described in Section 7.1 whose results are illustrated in Table 1.

(C2): Secpoline achieves similar overhead in syscall intensive applications compared to a state-of-the-art secure in-process syscall interposer and significantly lower overhead compared to a cross-process approach. This is proven in experiment (E2) described in Section 7.2 whose results are illustrated in Figure 3 and Figure 4.

(C3): Secpoline achieves competitive performance compared to a state-of-the-art MPK sandbox while avoiding the expensive cross-process fallback paths commonly required by process-isolated designs. This is proven in experiment (E3) described in Section 7.3 whose results are illustrated in Figure 5 and Figure 6.

(C4): Secpoline can transparently integrate large and complex existing codebases into its syscall interposition framework, enabling practical in-process implementations of real-world systems such as intrusion detection engines and proxy services. This is described in Section 8 and demonstrated by experiments (E4) and (E5), which integrate Falco and ProxySQL into Secpoline, respectively. In particular, (E4) shows that Secpoline can replace Falco's kernel-module-based syscall interception with a fully in-process interception approach, while (E5) demonstrates that Secpoline can virtualize inter-process proxy communication by embedding ProxySQL directly into the monitored process. The resulting systems maintain practical performance characteristics while supporting significantly more flexible in-process mediation and isolation. The results of these experiments are illustrated in Figure 7 and Figure 8.

A.4.2 Experiments

(E1): [Security Evaluation] [45 human-minutes + 30 compute-minutes]: Verification that Secpoline protects against a set of known attacks. This evaluation contains multiple proof-of-concept attacks. These should all crash with a `SIGSEGV`, `SIGILL`, or return an error code to demonstrate that the attack failed.
Preparation: Build Secpoline using CMake. This will build all proof-of-concept attacks, which can be found in `output/security_eval`. The README contains an explanation of how to set up the `Forged signal delivery` attack, as this requires some manual work.
Execution: Run each of the proof-of-concept attacks.
Results: All of these attacks should fail or terminate before completing. An attack succeeds if it outputs `"sud disabled"`.

(E2): [Nexpoline Benchmarks] [1 human-hour + 1 compute-hour]: Comparison of the overhead introduced by Secpoline, Nexpoline/Endokernel, and strace across several benchmarks, showing similar overhead between Secpoline and Nexpoline/Endokernel and significantly higher overhead for strace.
Preparation: Follow the instructions to set up the Endokernel Docker environment and update its test cases. These instructions can be found in `benchmarks/benchmark_endokernel.md`.
Execution: Run `zip.py`, `sqlite.py`, `curl.py`, and `lmbench.py`.
Results: Run `calc_overhead.py` for each of these tests. This will output the results shown in Figure 3 and Figure 4.

(E3): [Cerberus Benchmarks] [1 human-hour + 2 compute-hours]: Comparison against a state-of-the-art MPK sandbox across two HTTP servers.
Preparation: Obtain the Cerberus source code (https://github.com/ku-leuven-msec/The-Cerberus-Project) and the Apache HTTP Server source code (https://github.com/apache/httpd). Then build Cerberus, its benchmarks, and the

Apache HTTP Server.
Execution: Instructions to run both lighttpd and Apache can be found in `benchmarks/benchmark_cerberus.md`. This document contains the commands required to run the benchmarks under the different configurations described in the paper.
Results: The output of `wrk` shows the requests per second and reproduces the results shown in Figure 5 and Figure 6.

(E4): [Falco Case Study] [30 human-minutes + 15 compute-minutes]: Demonstration of Secpoline's ability to integrate large codebases and replace kernel modules for syscall interception without significant performance loss.
Preparation: Switch to the `falco_usecase` branch. After building Secpoline, update the `Secpoline_plugin` path in `falco/falco.yaml` and `falco/falco_base.yaml`.
Execution: Run the `zip_bench.sh` script in the `benchmarks/falco_benchmarks/` directory. This will install a kernel module to compare against the Secpoline interception approach.
Results: This outputs the total execution time in seconds for each run and should demonstrate the relatively minor performance cost of replacing a kernel module with Secpoline.

(E5): [ProxySQL Case Study] [30 human-minutes + 30 compute-minutes]: Demonstration of Secpoline's ability to integrate large codebases and achieve significant performance improvements through cross-process communication virtualization.
Preparation: Switch to the `proxysql_usecase` branch and build Secpoline. Ensure that `mysql` is installed. Then, in the `libproxysql/benchmark` directory, run the `init_mysql_database.sh` script, which sets up the database and user required by the benchmarks. The README also contains instructions for performing this step manually.
Execution: Run the `sysbench.sh` script.
Results: This outputs the total throughput and latency for each run.

A.5 Notes on Reusability

Secpoline is a highly customizable syscall interposer that can be easily extended to support a wide range of use cases. To adapt Secpoline for a new use case, add code to the following components:

Adding new syscall filter rules. To add new filter rules, define a new handler function in `main_monitor.cpp` using the `SYSCALL_HANDLER` macro and declare it in `secpoline.h` using the `DECLARE_HANDLER` macro. These handlers use the `gprs` class to read from and write to application registers. The functions `memcpy_untrusted_to_trusted()` and `memcpy_trusted_to_untrusted()` provide safe access to untrusted memory, which is otherwise disabled by the exclusive MPK permission policy.

Syscall handlers should always return `false`.

Assigning handlers to syscalls. Once a new handler has been defined, it must be assigned to a syscall in `secpoline.cpp:initialise_syscall_handlers()`. There are three types of handlers:

- **Prehandlers:** Any number of prehandlers can be added for a syscall. These execute before the main handler and should not invoke the syscall on behalf of the application. Instead, they may modify syscall arguments before execution.
- **Main handler:** Only one main handler can be assigned to a syscall. While prehandlers and posthandlers may invoke syscalls internally, only the main handler should emulate the syscall for the application.
- **Posthandlers:** Any number of posthandlers can be added for a syscall. These execute after the main handler and should assume that the syscall return value is already stored in `rax`.

Initialization code. Add any initialization code to `secpoline.cpp:main()` after the call to `init_page_manager()`, as this is the point where Secpoline is fully initialized and ready to support arbitrary code.

Meta-monitor. To add new syscall filter rules for monitoring syscalls, update `meta_monitor.cpp:meta_monitor()`. Do not call any monitor-unaware code from this function, and ensure that all syscalls executed during meta-monitor handling are inlined to avoid replacement with `call *rax`.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2026/.