Practical Data-Only Attack Generation

Brian Johannesmeyer, Asia Slowinska, Herbert Bos, and
Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

https://www.usenix.org/conference/usenixsecurity24/presentation/johannesmeyer

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

USENIX Security '24 Artifact Appendix: Practical Data-Only Attack Generation

Brian Johannesmeyer Asia Slowinska Herbert Bos Cristiano Giuffrida

Vrije Universiteit Amsterdam

A Artifact Appendix

A.1 Abstract

In this artifact, we provide the means to reproduce our main results. Specifically, we show that our exploitation pipeline, EINSTEIN, identifies vulnerable syscalls across a range of applications, and that it generates working data-only exploits against `nginx`. We have evaluated our artifact using an AMD Ryzen 9 3950X CPU (32 cores), with 128GB of RAM, 4 TB storage, and running Ubuntu 22.04.3 LTS (kernel v6.2). Our source code is available on GitHub[1].

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Although EINSTEIN indeed produces working exploits, they are non-destructive proof-of-concept exploits, which write the string `"HELLO"` to either a file (`"/tmp/hi"`) or a local socket (address `"192.0.2.0"`). Hence, evaluating EINSTEIN poses no risks for machine security, data privacy, or other ethical concerns.

A.2.2 How to access

The files for the artifact evaluation are available at the `ae` tag of the EINSTEIN repository[2].

A.2.3 Hardware dependencies

EINSTEIN requires an x86-64 machine (Intel recommended); enough RAM to simultaneously load multiple program snapshots into memory, so EINSTEIN can post-process reports in parallel (recommended 100 GB RAM); and enough storage for hundreds of program snapshots (minimum 2 TB storage for this evaluation). We recommend using a machine with a high core count to speed up EINSTEIN's report post-processing.

A.2.4 Software dependencies

To build EINSTEIN and the target programs, we expect certain packages to be installed. In the Section A.3, we detail the steps to install such dependencies on Ubuntu 22.04, but similar steps are needed for other distributions.

A.2.5 Benchmarks

We use each target application's test suite to drive the analysis.

A.3 Set-up

To download and install dependencies, including go-task as a task-runner, from the EINSTEIN repository, run: `sudo snap install task --classic && task init`.

A.3.1 Installation

To build `libdft`[3], the command server, the EINSTEIN tool, and all target applications, run: `task libdft-build cmdsvr-build einstein-build apps-build`.

A.3.2 Basic Test

We first make a couple notes about running EINSTEIN:

- Due to the non-deterministic nature of dynamic analysis (from concurrency issues, system variability, etc.)[4], the actual results may slightly deviate from the expected results.
- If the `db-analyze-reports` task fails, try running the `db-analyze-reports-singleproc` task instead. It will be slower, but will avoid any system load-related crashes.

Test that the different components work as follows:

(T1): *libdft memory tainting [1 compute-second].*
To test libdft's "taint all memory" functionality, run `task libdft-test -- memtaint` *and compare its output to the expected output.*

[1] https://github.com/vusec/einstein/
[2] https://github.com/vusec/einstein/releases/tag/ae
[3] https://github.com/vusec/libdft64-ng
[4] See "Deterministic Process Groups in dOS" (OSDI 2010) and "Node.fz: Fuzzing the Server-Side Event-Driven Architecture" (EuroSys 2017).

(T2): *libdft instruction tainting [1 compute-second].*
To test libdft's per-instruction taint policies, run `task libdft-test -- ins` *and compare its output to the expected output.*

(T3): EINSTEIN *tool [1 compute-minute].*
To test EINSTEIN *on a simple program, run* `task einstein-test`*. Then, compare the output of* `task db-print-candidates` *with the expected output.*

(T4): *Target applications [4 compute-minutes].*
To test EINSTEIN *running each target application with a simple workload (e.g., sending a simple GET request to a web server), run* `task reports-clean apps-test db-add-reports db-analyze-reports`*. Then, compare the output of* `task db-print-candidates` *with the expected output.*

(T5): *Target application test suites [20 compute-minutes].*
To test EINSTEIN *running each target applications' test suites for 2 minutes each (rather than the entire test suites), run* `task reports-clean apps-eval-brief db-add-reports db-analyze-reports`*. Then, compare the output of* `task db-print-candidates` *with the expected output.*

(T6): *Exploit confirmation [2 compute-minutes].*
To test EINSTEIN*'s exploit confirmation for* `nginx`*, run* `task reports-clean einstein-nowrite-config nginx-eval-custom db-add-reports db-analyze-reports db-analyze-candidates`*. Then, compare the output of* `task db-print-exploits` *with the expected output.*

A.4 Evaluation workflow

A.4.1 Major Claims

We make the following claims:

(C1): EINSTEIN *identifies thousands of vulnerable syscalls in common server applications. This is proven by Experiment (E1).*

(C2): EINSTEIN *generates hundreds of working exploits against* `nginx`*. This is proven by Experiment (E2).*

A.4.2 Experiments

We prove the above claims using the following experiments:

(E1): *Vulnerable syscall identification [24 compute-hours].*
How to: *We will run each application with* EINSTEIN*, then analyze the reports to identify vulnerable syscalls.*
Preparation: *Run* `task reports-clean` *to remove past reports.*
Execution: *Run* `task apps-eval db-add-reports db-analyze-reports`*.*
Results: *Compare the output of* `task db-print-candidates` *to the expected output. The output contains thousands of vulnerable gadgets, broken down by: (i) syscall and argument type (i.e., Table 3), and (ii) target application (i.e., Table 4)—thereby proving Claim (C1).*

(E2): *Exploit generation [12 compute-hours].*
How to: *We will run* `nginx` *with* EINSTEIN*, then analyze the reports to identify vulnerable syscalls, then confirm candidate exploits as working exploits.*
Preparation: *Run* `task reports-clean` *to remove past reports.*
Execution: *Run* `task nginx-eval db-add-reports db-analyze-reports db-analyze-candidates`*.*
Results: *Compare the output of* `task db-print-exploits` *to the expected output. The output contains hundreds of confirmed exploits for* `nginx` *(i.e., Table 5)—thereby proving Claim (C2).*

A.5 Notes on Reusability

This prototype may be expanded in a few directions:

- To modify EINSTEIN's taint policies (e.g, to target more syscalls, or to target syscall-guard variables), modify the EINSTEIN tool in `src/einstein`.
- To run the target applications (e.g., `nginx`) with other workloads, first start the application with EINSTEIN (`cd apps/nginx-1.23.0 && RUN_EINSTEIN=1 ./serverctl restart`), then run the custom workload (e.g., `echo 'Hello!' | netcat 127.0.0.1 1080`).
- To run EINSTEIN on other applications:
  1. Add the application to the `apps/` directory;
  2. Copy the files `serverctl` and `clientctl` from another application's directory into its directory, and modify them to start the application's server and a client for it; and
  3. Ensure that the application's build script generates position-independent code (i.e., the default on most compilers).
- To write another Pin tool that uses `libdft`:
  1. Copy the EINSTEIN tool, e.g.: `cp -r src/einstein src/my-tool`;
  2. Modify `MY_TOOL` and `MY_OBJS` in the `Makefile`;
  3. Modify the source code to suite your analysis;
  4. Build it: `cd src/my-tool && -DLIBDFT_TAG_PTR -DLIBDFT_PTR_32 -DLIBDFT_TAG_SSET_MAX=16' make obj-intel64/my-tool.so`; and
  5. Run it on some target application: `setarch x86_64 -R ./src/misc/pin-3.28-98749-g6643ecee5-gcc-linux/pin -t src/my-tool/obj-intel64/my-tool.so -- echo 'Hello!'`.

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/usenixsec2024/.