USENIX Security '24 Artifact Appendix: SafeFetch: Practical Double-Fetch Protectionwith Kernel-Fetch Caching

Victor Duta Mitchel Josephus Aloserij Cristiano Giuffrida

Vrije Universiteit Amsterdam

A Artifact Appendix

A.1 Abstract

The `SafeFetch` artifact can be used to reproduce the main claims in our paper as follows: a) `SafeFetch` mechanics can thwart kernel double-fetches: we provide a security workflow showing that `SafeFetch` can defend against a publicly available POC for a known double-fetch CVE; b) `SafeFetch` provides comprehensive security with low overheads: we supply a performance measuring workflow that shows `SafeFetch` incurs practical overheads across a series of benchmarks (i.e., LMBench, OSBench, Phoronix); c) `SafeFetch` achieves comprehensive security at a fraction of the performance cost of state-of-the-art solutions: which we prove by running the same performance workflow with Midas.

A.2 Description & Requirements

The artifact for `SafeFetch` is available on Github and contains the following components: a) the code implementing `SafeFetch` on the Linux kernel (v5.11); b) precompiled kernel images used to obtain the main security and performance results outlined in the paper; c) the raw performance results shown in the paper; d) scripts to reproduce the main performance and security results and e) detailed information regarding the artifact workflow. The artifact workflow for performance evaluation should be run on real-hardware for accurate performance results. Similarly, as the POC used in the security workflow exploits timing differences between concurrent writes and kernel reads, the security workflow should also be run on bare-metal.

A.2.1 Security, privacy, and ethical concerns

The CVE exploited in our evaluation also contains the fix, as such, host machines are not exposed to any security threats while running the artifact workflow. However, the artifact will remove prior Phoronix results from the host machine.

A.2.2 How to access

Our artifact is available on GitHub.

- Repository: https://github.com/vusec/safefetch-ae
- Release: v1.0

Steps to access the stable source code for the `SafeFetch` prototype on Linux Kernel (v5.11) are explained in the aforementioned repository readme.

A.2.3 Hardware dependencies

The precompiled kernel images used to reproduce the main results in the paper should be loaded on a real machine for accurate performance tests. The host machine requires around 40 GiB free disk space, and at least 8GiB of memory. Our `SafeFetch` prototype supports machines with 64-bit x86 processors, and the results in the paper were obtained on a Intel i7-6700 machine (using 32 GiB of RAM). An SSD is prefered for storage as it leads to faster compilation should you need to re-compile the workflow kernels. Moreover, for filesystem benchmarks (e.g., ran during LMBench bandwidth tests) an SSD assures that the storage speed does not become the bottleneck for the experiments.

A.2.4 Software dependencies

Running the `SafeFetch` images requires a guest operating system which supports GRUB 2.0 (or newer), as the artifact scripts use GRUB utilities to load the precompiled kernels on the host machine (check whether the `grub-mkconfig` and `update-grub2` are available on the machine). The images were tested on a machine running `Ubuntu 22.04` with a host Linux kernel version `5.15.0-100-generic`. The precompiled kernels use the default `X86_64` Linux build, `x86_64_defconfig`. In some scenarios the kernels might not run on the host machine (e.g., the host machine is equipped with hardware which require drivers beyond the Linux default build). In these cases, you must compile the kernels locally on the machine, using the host kernel local config. The artifact provides scripts to automatically compile `SafeFetch` kernels on the host machine. For local compilation, the host machine ideally runs gcc `v8.4` (or newer) and binutils `v2.30` (or newer). The process is explained in detail in the readme files that come with the artifact.

A.2.5 Benchmarks

None. All the benchmarks are automatically installed as part of the artifact.

A.3 Set-up

A.3.1 Installation

You can download our stable release:

```
$ wget https://github.com/vusec/
    safefetch-ae/archive/refs/tags/v1
    .0.zip -O safefetch-ae.zip
$ unzip -a safefetch-ae.zip
```

Or clone the artifact from the GitHub repository:

```
$ git clone
    https://github.com/vusec/safefetch-ae.git
```

Once the artifact is saved on the host machine, enter the artifact folder (i.e., `safefetch-ae`) and run the following command:

```
$ make setup
```

The command will install all dependencies required by our workflow scripts on the host machine. Furthermore it will install and configure our main performance benchmarks: LMBench, OSBench and the Phoronix benchmarks discussed in the paper. Lastly, it will download the publicly available POC for `CVE-2016-6516` used to evaluate `SafeFetch`'s security.

A.3.2 Basic Test

After setup finishes first check whether the POC and benchmarks are installed correctly and you can generate the main artifact results:

```
$ make verify_install && make all-
    paper
```

The end result of the command will be a pdf document in the `./playground/paper/` directory (which we will later regenerate once more parts of the artifact are ran). Without running the rest of the evaluation workflow the pdf will only contain the main performance tables/graphs discussed in the `SafeFetch` performance evaluation (obtained from the RAW results that are included with the artifact) alongside boiler plates for the results obtained when running the next parts of the artifact workflow. The backbone of the artifact is loading different kernels and running different performance/security evaluation workflows. To finish the basic test, check for liveness on the kernel used to reproduce the security workflow.

```
$ make load_kernel SAVED_DIR=exploit-
    default
$ sudo reboot
```

When loading the kernel you will be prompted to chose which kernel to load by default after reboot. Pick the index of the first kernel containing the string id "5.11.0-exploit+". In case the default `exploit` kernel does not boot, follow the detailed explanation provided in the artifact repository readme, which explains how to compile this kernel on the host machine.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *`SafeFetch`'s caching mechanism structurally mitigates double-fetch bugs in the Linux Kernel. This is proven by experiment (E1) which is described in Section 9.1 of the main paper. The experiment shows that even though the publicly available POC (proof-of-concept) to exploit a known double-fetch CVE can succesfully execute on a baseline image which does not enforce `SafeFetch`'s mechanics, the bug cannot be reproduced on a kernel that enables `SafeFetch`.*

(C2): *`SafeFetch` mitigates double fetch bugs with small penalty to performance. This is proven by experiment (E2) and described in our main performance evaluation section (Section 9.2) which shows that the default `SafeFetch` configuration scores low performance overheads (relative to the baseline), tipically in a 5% performance budget, across a series of microbenchmarks (e.g., LMBench, OSBench) and real-world applications (from the Phoronix suite).*

(C3): *`SafeFetch`'s lightweight caching mechanism achieves comprehensive performance at a fraction of the cost of existing solutions (e.g., Midas). This is proven by experiment (E3) and highlighted by our performance evaluation (Section 9.2) which shows that the overhead of Midas is far larger than the penalty incurred by `SafeFetch` on various benchmarks (e.g., 35.9% geo mean for Midas vs 4.4% for `SafeFetch` on LMBench).*

A.4.2 Experiments

(E1): *[Security artifact] [1 human-minutes + 40 compute-minutes]: `SafeFetch` protects the kernel against double-fetch bugs and in particular mitigates an exploit for `CVE-2016-6516`. For this experiment you will execute the POC reproducing the exploit with and without `SafeFetch` enabled.*

How to: We provide a precompiled kernel image for this experiment. After booting, the image will act as baseline (the `SafeFetch` mechanism is disabled using static keys, i.e., the `SafeFetch` hooks are nop instructions). During runtime, the `SafeFetch` defense can be enabled using a script provided with the artifact (by enabling the static keys). If the bug reproduces during the execution of

the POC, a message containing the string "Bug-Warning" is printed to dmesg output. The experiment runs the POC for 5 iterations, each iteration attempting to reproduce the bug 1 million times and collects the number of warnings printed during each iteration. Our workflow script will run the experiment first while the kernel runs as baseline, after which it enables `SafeFetch` and runs the experiment again. While the kernel runs as baseline it's expected that the bugs reproduces a dozen of times per iteration. When `SafeFetch` is enabled, no warning should be printed to dmesg.

Preparation: From the artifact root dir run the following command to load the kernel and reboot the machine:

```
$ make load_kernel SAVED_DIR
    =exploit-default
$ sudo reboot
```

Execution: After booting into the new kernel, from the artifact root run the following command:

```
$ make run_security_artifact
```

As a fail-safe, this command only runs within the exploit kernel.

Results: After the experiment finishes, regenerate the artifact pdf by running:

```
$ make all-paper
```

Section 2 of the pdf will now contain tables showing how many times the bug was reproduced for each iteration, on the two configurations.

(E2): *[Performance artifact for `SafeFetch`] [1 human-minutes + 7 compute-hours + 5GB disk]: `SafeFetch` demonstrates effective double-fetch mitigation with low overhead. For this experiment you will run the baseline and `SafeFetch` default configuration (refered to as `SafeFetch-default` in the main paper) across multiple benchmarks: LMBench (latency and bandwidth), OSBench and Phoronix.*

How to: We provide a image for this experiment configured with static keys to run first as a baseline (with the defense hooks replaced with nop instructions) then with the `SafeFetch` default configuration enabled.

Preparation: From the artifact root dir run the following command to load the kernel and reboot the machine:

```
$ make load_kernel SAVED_DIR
    =safefetch-default
$ sudo reboot
```

When prompted for which kernel to boot, provide the index of the first image containing the string id "5.11.0-safefetch+".

Execution: After booting into the new kernel, from the artifact root run the following command:

```
$ make
    run_performance_artifact
```

The script will first run all benchmarks on the baseline then switch to the `SafeFetch` configuration and run the same benchmarks again. RAW results are outputed in the `./playground/performance` directory.

Results: After the experiment finishes, regenerate the artifact pdf by running:

```
$ make all-paper
```

The pdf now contains all results for `SafeFetch-default` in Section 1, each in a different subsection (shown side-by-side with the results from the main paper). Expect the following trends in the obtained results:

- LMBench: geomean overheads around 4-5% for both syscall and bandwidth tests.
- OSBench: geomean overhead between 0-3%.
- Phoronix: git, pybench, openssl oveheads between 0-1%.
- Phoronix: apache and nginx overheads around 2-3%.
- Phoronix: around 7% overhead for the IPC benchmark.

(E3): *[Performance comparisson with midas] [1 human-minutes + 3.5 compute-hours + 5GB disk]: `SafeFetch` demonstrates effective double-fetch mitigation with lower overheads than the state-of-the-art. For this experiment you will run Midas across the same performance benchmarks and compare with `SafeFetch-default`.*

Preparation: From the artifact root dir run the following command to load the kernel and reboot the machine:

```
$ make load_kernel SAVED_DIR
    =midas-default
$ sudo reboot
```

When prompted for which kernel to boot, provide the index of the first image containing the string id "5.11.0-midas+".

Execution: After booting into the new kernel, from the artifact root run the following command:

```
$ make
    run_performance_artifact
```

Results: After the experiment finishes, regenerate the artifact pdf by running:

```
$ make all-paper
```

The pdf will now contain the results for Midas in the same performance tables/graphs as `SafeFetch`. Expect the following trends in the obtained results:

- higher overheads for Midas for LMBench latency results and OSBench (relative to `SafeFetch`).
- Phoronix: similar results as `SafeFetch` for git, pybench, openssl

- Phoronix: higher results for nginx and apache for Midas (around 7% on nginx and as high as 14% for apache).

Though not often, the Midas kernel can crash, disturbing the flow of the artifact. As a fail-safe the paper generation process can generate partial results from unfinished benchmarking runs.

A.5 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/.