PET: Prevent Discovered Errors from Being Triggered in the Linux Kernel

Zicheng Wang, *Nanjing University;* Yueqi Chen, *University of Colorado Boulder;* Qingkai Zeng, *Nanjing University*

https://www.usenix.org/conference/usenixsecurity23/presentation/wang-zicheng

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

USENIX'23 Artifact Appendix: <PET: Prevent Discovered Errors from Being Triggered in the Linux Kernel>

Zicheng Wang*
wzc@smail.nju.edu.cn
Nanjing University

Yueqi Chen
yueqi.chen@colorado.edu
University of Colorado Boulder

Qingkai Zeng
zqk@nju.edu.cn
Nanjing University

A Artifact Appendix

A.1 Abstract

This artifact is applying for an **Artifacts Available** badge, an **Artifacts Functional** badge, and an **Results Reproduced** badge. It provides two main artifact sets for evaluators to reproduce `PET`. The first artifact set, detailed in Github, enables constructing `PET` from scratch, and the second artifact set includes kernel images and a root filesystem which allow the evaluator to reproduce our results in an isolated environment without any destructive steps.

Both artifact sets include Proof of Concept (PoC) programs and exploits of vulnerabilities used as test cases and eBPF programs that enable `PET` protection. After installing the eBPF program, the evaluator can execute the PoC programs and exploits to access the effectiveness of `PET`, by observing that the error triggering is prevented.

Besides we include user guidance in the first artifact set to help readers understand the design of `PET`, and develop their own eBPF programs for more error types that have not been covered in `PET` so far.

In this appendix, we will provide necessary instructions for evaluators to reproduce `PET` as well as an example along with screenshots for illustration.

A.2 Description & Requirements

In this section, we first describe whether reproducing our artifacts will risk the evaluator's machine security, followed by approaches to accessing our artifacts. Then, we describe hardware dependencies and software dependencies before listing the benchmarks.

A.2.1 Security, privacy, and ethical concerns

`PET` aims to protect the OS kernel through the eBPF ecosystem. To enable `PET`, the kernel needs additional eBPF helper functions before being compiled and installed, which is destructive to some extent. Therefore, to make evaluators feel safe, we prepared a kernel image and a root filesystem for evaluators. As such, evaluators can download the image and reproduce our results in an isolated environment, ensuring the safety and privacy of the host machine. The access for the image can be found in § A.2.2.

Furthermore, it is important to note that all vulnerabilities and proof-of-concept programs included in the artifact are publicly available and have been addressed in the mainstream kernel. Therefore, there are no security, privacy, or ethical concerns regarding the open-source community. The artifact builds upon resolved issues, and its purpose is to contribute to the knowledge and advancement of the field.

A.2.2 How to access

The complete artifacts are available in a public Github Repo `https://github.com/purplewall1206/PET`, which includes three main components: eBPF programs and corresponding scripts for evaluation, source code developed in `PET`, manuals and examples for evaluators to quickly understand the key idea of `PET`. Due to the space limit, we cannot list all details from building kernel to compiling eBPF programs to enable `PET` protections. Therefore, the repo also includes elaborate instructions for evaluators to follow.

The artifacts provided in the Github Repo are sufficient for evaluators to reproduce. However, as we mentioned in § A.2.1, the reproducing procedure includes destructive steps. To this end, we additionally provide a root filesystem and a compiled kernel image, which are in `https://tinyurl.com/2428uac5`.

A.2.3 Hardware dependencies

To completely reproduce `PET`, we recommend the following minimum hardware configurations: ❶ an Intel CPU with VT-X virtualization feature, ❷ 8GB or larger memory, and ❸ at least 100GB disk space.

A.2.4 Software dependencies

It is preferable to perform the evaluation on the Ubuntu Linux distro, especially the 20.04 desktop which is the same OS for `PET` development. The OS is supposed to include essential packages such as `debootstrap`, `qemu-system-x86_64`, `open-ssh`,

*The work was done while visiting the University of Colorado Boulder.

and `wget`. These packages are necessary for setting up the evaluation environment and conducting runtime evaluations.

Besides, it is advised **not** to utilize Docker for the evaluation process because the artifact necessitates the use of two separate terminals - one for executing the Proof of Concept programs and another for displaying the output of the eBPF programs.

A.2.5 Benchmarks

The Proof of Concept programs for vulnerabilities used as test cases have been collected and provided in the Github Repo. We used Phoronix-benchmark for performance measurement which is publicly available online.

A.3 Set-up

In this section, we focus on the installation and testing of `PET` using the kernel image and a root filesystem we provided for the sake of ethics (§ A.2.2). The evaluator can boot up the kernel using `QEMU`. Due to the space limit, we move the detailed instruction for reproducing `PET` from scratch in the Github Repo.

A.3.1 Functional

For the functional evaluations, we have implemented a `evaluate.sh` script to set up the environments. including: ❶ use `apt` to install required software mentioned in A.2.4, ❷ pop up 2 terminals, *terminal 1* start the virtual machine, and *terminal 2* connect to the virtual machine with `ssh`, and ❸ copy the test scripts into virtual machine.

A.3.2 Reproduce

For the reproducible evaluations, we first present a `phoro-run.sh` script to reproduce all performance results. We also present an instruction for generating new BPF prevention programs from scratch and evaluate it in the also in the functional testing environment.

A.3.3 Installation

None.

A.3.4 Basic Test

After evaluators pull the github repository and run `evaluate.sh`, there will be 2 terminals pop up. In figure 1, the left *terminal 1* boots up the virtual machine, and waits to be logged in, and the right *terminal 2* has already logged in through `ssh`. Evaluators can login the virtual machine with the user name `root` and no password is needed. After that, evaluators can run `ls` command on either terminal, and there will be three directories, including `bpf`, `PoCs`, `scripts`.

The `bpf` directory contains all compiled BPF prevention programs. The `POCs` directory contains all compiled proof-of-concepts programs that can trigger the vulnerabilities. The `scripts` directory contains evaluation scripts that need evaluator to execute in the virtual machine.

A.4 Evaluation workflow

As described in Section 5 (Error-dependent Prevention Policies) of our paper, the PET framework provides support for preventing five distinct types of errors from being triggered. To evaluate the functional of each type of kernel error prevention, the artifact includes five BPF prevention programs.

During the evaluation process, the virtual machine will initiate the execution of these five BPF protection programs immediately after boot-up. Subsequently, the evaluator can proceed to run the proof-of-concept tests for the corresponding five vulnerabilities. The BPF prevention programs are designed to intercept and bypass the error-prone sections within the kernel. As a result, the system will continue to function smoothly, ensuring the stability and integrity of its operations.

After the functional evaluation, evaluators can also reproduce the performance overhead, and try to add new BPF programs to prevent the other vulnerabilities from being triggered.

A.4.1 Major Claims

C1: The 5 types of kernel errors (integer overflow, out-of-bound, use-after-free, uninitialization, data race) will be prevented. **C2:** The system will keep functioning after the errors are prevented from being triggered.

A.4.2 Experiments

Functional: First of all, evaluators need to change directories to `/root/scripts` in both terminals, then execute `start-bpf-progs.sh` to start all 5 BPF programs in the *terminal 2*. After that, output of the 5 BPF programs will be printed in the *terminal 2*. We have also prepared a video to demonstrate the evaluation workflow on Youtube https://www.youtube.com/watch?v=0BVsULXT0xI.

(E1): Test if the BPF program can prevent an integer overflow vulnerability CVE-2017-7184 from being triggered.
Execution: execute `test-CVE-2017-7184.sh` in *terminal 1*
Results: There will be `killed` signal in *terminal 1*, and there will be a report `====CVE-2017-7184 is happened====` in *terminal 2*.

(E2): Test if the BPF program can prevent an out-of-bound vulnerability CVE-2016-6187 from being triggered.
Execution: execute `test-CVE-2016-6187.sh` in *terminal 1*
Results: There will be `killed` signal in *terminal 1*, and there will be a report `====CVE-2016-6187 is happened====` in *terminal 2*.

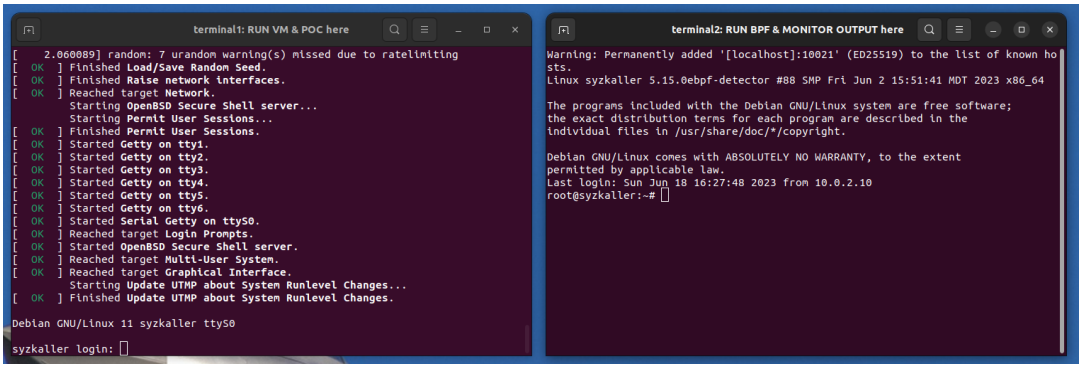

Figure 1: Gnome terminals, *terminal 1* boot up the virtual machine, *terminal 2* connect to the virtual machine through `ssh`.

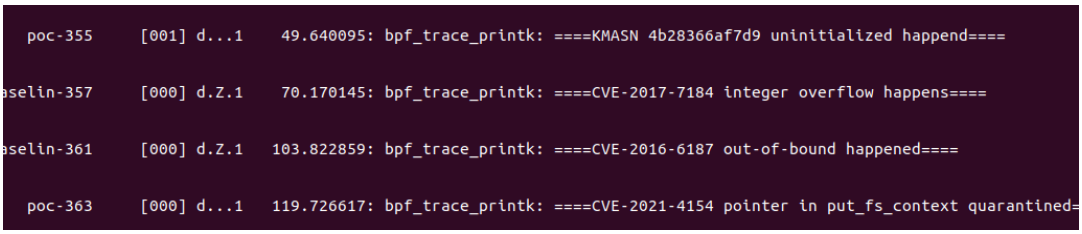

Figure 2: Outputs of the BPF prevention programs in *terminal 2*.

(E3): Test if the BPF program can prevent an use-after-free vulnerability CVE-2021-4154 from being triggered.
Execution: execute `test-CVE-2021-4154.sh` in *terminal 1*
Results: Because of the quarantine & sweep policy, the dangling pointer of a use-after-free vulnerability will be quarantined, the vulnerability will not be triggered. Proof-of-concept in *terminal 1* cannot trigger the vulnerability, and *terminal 2* will report dangling pointers are quarantined.

(E4): Test if the BPF program can prevent an uninitialization vulnerability kmsan-4b28366af7d9 from being triggered.
Execution: execute `test-kmsan_4b28366af7d9.sh` in *terminal 1*
Results: The vulnerability is triggered in *terminal 1* under an conservative check policy, it means that the BPF program will catch the uninitialization memory but not kill the process. There will be reports `====kmsan-4b28366af7d9 is happened====` in *terminal 2*.

(E5): Test if the BPF program can prevent a data race vulnerability kcsan-dcf8e5633e2e from being triggered.
Execution: KCSAN does not provide proof-of-concept, so no proof-of-concept is executed in *terminal 1*.
Results: The BPF program `detector_kcsan_dcf8e5633e2e` will be keep checking if the vulnerability kcsan-dcf8e5633e2e is being triggered.

The evaluation results, as depicted in Figure 2, demonstrate the successful prevention of various vulnerabilities. Starting from the top, the artifact effectively prevent uninitialized variables, integer overflows, and out-of-bound vulnerabilities. Additionally, the artifact identifies and quarantines the potential dangling pointers of use-after-free vulnerabilities. It is important to note that evaluators have the freedom to execute additional commands on *terminal 1* during the evaluation process. Despite the execution of these commands, the system remains functional and unaffected by the errors.

Reproducible The reproducible evaluation includes 2 part, the performance test and add new BPF prevention programs.

(E1) : test performance overhead when BPF prevention programs protect the system.
Execution: (about 2 hour for each performance test, 14 hours in total) execute the `phoro-run.sh` scripts in the *terminal 1*, and 7 seperate performance tests are queued to be executed, including the vanilla, system protected by 5 BPF program individually and simultaneously.
Results: execute `phoronix-benchmark ↪ start-result-viewer`

(E2) : add new BPF program to prevent new
Execution: (about 10 minutes) We present an instruction including a demo(a use-after-free), executor can follow the guidance to extract the sanitizer report and generate a new program. Similar to the functional evaluation, evaluators can also evaluate the effectiveness of the new added BPF program.
Results: After execute proof-of-concept in *terminal 1*, there will be report that dangling pointers are quarantined in *terminal 2*.

The results show that artifact can reproduce the performance overhead and easy to add new BPF programs for upcoming kernel vulnerabilities.