PUMM: Preventing Use-After-Free Using Execution Unit Partitioning

Carter Yagemann, *The Ohio State University;* Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee, *Georgia Institute of Technology*

https://www.usenix.org/conference/usenixsecurity23/presentation/yagemann

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: PUMM: Preventing Use-After-Free Using Execution Unit Partitioning

Carter Yagemann
The Ohio State University*

Simon P. Chung
Georgia Institute of Technology

Brendan Saltaformaggio
Georgia Institute of Technology

Wenke Lee
Georgia Institute of Technology

A Artifact Appendix

A.1 Abstract

The artifact is a code repository (with supporting documentation) for PUMM, a runtime Linux defense that prevents use-after-free vulnerabilities from being exploitable. PUMM consists of an offline profiling phase that generates a security profile for an online monitor that wraps libc's memory management functions (e.g., malloc, free, etc.).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact should not pose any inherent security, privacy, or ethical concerns. No preexisting data is read or transmitted and no preexisting defenses are disabled or bypassed. If the user decides to install software containing use-after-free vulnerabilities for the purposes of verifying PUMM's security claims, they do so at their own risk. For security and ethical reasons, the artifact does not include any vulnerable programs.

A.2.2 How to access

The artifact is a code repository and documentation that can be accessed on Github: https://github.com/carter-yagemann/PUMM/tree/91e58cd5d929e25d0b83fdfd0ec3c5517e2a32e7.

A.2.3 Hardware dependencies

PUMM requires a *baremetal* Linux computer (virtual machines are not supported) running an Intel Core processor (e.g., i3, i5, i7, etc.).

A.2.4 Software dependencies

The preferred environment for running PUMM is Debian Buster, however PUMM should work with any recent version of Debian or Ubuntu. PUMM has several software dependencies, including cmake, gawk, Graph-Tool, and Linux Perf. Users should follow the Setup section in the README.

A.2.5 Benchmarks

The primary benchmark used in the paper is a collection of open source programs with known use-after-free vulnerabilities and publicly reported steps for triggering these issues. Reference IDs are provided in Table 1 of the paper and supporting artifacts can be located by looking up the CVE in the National Vulnerability Database[1] or the issue number in the affected program's bug tracking website.

Additionally, performance and memory overheads are measured using the SPEC CPU 2006 standard benchmark. FFmalloc and MarkUs are used as baseline defenses for comparison.

A.3 Set-up

A.3.1 Installation

Users should follow the Setup section of the README.

A.3.2 Basic Test

Users should follow the Usage section of the README, which will cover all core components of PUMM, using /bin/ls as the target program to be protected:

1. Collecting runtime traces of the target program.
2. Generating a security profile for the target program.
3. Using PUMM's runtime monitor (and generated profile) to protect the target program during execution.

The README provides example terminal outputs for each step for users to verify success. Notice that because the basic test does not involve launching an exploit, the expected outcome of using the online portion of PUMM is no observable change to the target program's behavior.

*Work done while at the Georgia Institute of Technology.

[1] https://nvd.nist.gov/

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): PUMM is able to prevent 40 real-world use-after-free exploits targeting 26 popular open source programs. This is proven by experiment (E1) described in Section 4.1 of the paper and illustrated in Table 2.

(C2): PUMM prevents the same vulnerabilities as FFmalloc and MarkUs while aborting in fewer cases. This is proven by experiment (E2) described in Section 4.1 of the paper and illustrated in Table 3.

(C3): PUMM incurs 2.74% less performance overhead than FFmalloc and 52.0% less memory overhead than MarkUs on the SPEC CPU 2006 standard benchmark. This is proven in experiment (E3) described in Section 4.2 of the paper and illustrated in Figures 4 and 5.

(C4): PUMM is able to prevent use-after-free exploitation in all but 4 cases out of 3,000 synthetically generated vulnerabilities. This is proven by experiment (E4) described in Section 4.3 of the paper and illustrated in Figures 8, 9, and 10.

A.4.2 Experiments

(E1): [5 human-days + 3 compute-days + 300GB storage]: Collected vulnerable programs and triggering inputs, record traces, generate profiles, and verify that the vulnerability is triggered without PUMM's defense and then prevented with PUMM.
Preparation: For each vulnerability in Figure 2 of the paper, the affected software must be downloaded and compiled. A triggering input for the program must also be downloaded and verified to be working (e.g., by causing a segmentation fault).
Execution: PUMM should be used to generate a security profile for the target program, using the recorded traces.
Results: Running the target program with the triggering input and PUMM's online monitor activated should prevent the use-after-free from being exploitable. Notice that the target program may still crash, so verification requires debugging.

(E2): [4 human-hours + 30 compute-minutes + 1GB storage]: Collected vulnerable programs and triggering inputs should be executed with PUMM, MarkUs, and FFmalloc to observe whether the vulnerability is exploitable.
Preparation: Download and compile FFmalloc and MarkUs. Dataset preparation is covered in E1.
Execution: The vulnerable program should be executed with the triggering input for each defense.
Results: Running the target program with the triggering input and PUMM's online monitor activated should prevent the use-after-free from being exploitable. Notice that the target program may still crash, so verification requires debugging.

(E3): [2 human-hours + 2 compute-hours + 50GB storage]: Compare the runtime and memory overheads of PUMM, FFmalloc, and MarkUs using the SPEC CPU 2006 benchmark.
Preparation: Download and compile SPEC CPU 2006.
Execution: Run the benchmark with no evaluated defenses enabled to establish a baseline. Then rerun the benchmark with PUMM, FFmalloc, and MarkUs enabled and calculate overheads.
Results: PUMM should have a lower average runtime and storage overhead than FFmalloc and MarkUs.

(E4): [1 human-day + 2 compute-hours + 2GB storage]: Synthetically generate use-after-free vulnerabilities using the monkey scripts provided in PUMM's code repository and verify whether PUMM prevents their exploitation.
Preparation: Compile the vulnerable programs described in E1.
Execution: For each vulnerable program, run the monkey script provided in the Scripts directory of the code repository, with and without PUMM's runtime monitor enabled.
Results: Without PUMM, a subset of the vulnerabilities generated with the monkey script should yield observable behaviors like segmentation faults. With PUMM, there should be fewer of these adverse behaviors.

A.5 Notes on Reusability

The scripts directory of the code repository contains scripts for using and evaluating PUMM. The scripts and their intended purposes are as follows:

build.sh : Build script to help compile PUMM. See the README in the code repository for full build steps.

dump-vdso.py : Helper script invoked by trace.sh to copy the system's vDSO object. Users should not need to call this script directly.

hook-debug.sh : Runs a target program with PUMM's protection activated. Uses a debug build of PUMM's runtime hooks with extra verbosity.

hook-monkey.sh : Runs a target program with PUMM's protection activated and additional code to synthetically inject use-after-free vulnerabilities for evaluation. Users should use monkey.sh instead of calling this script directly.

hook.sh : Runs a target program with PUMM's protection activated. Uses the optimized production build of PUMM's runtime hooks.

monkey-debug.sh : Runs a target program with PUMM's protection activated and synthetically injects a use-after-free vulnerability for evaluation. Whereas monkey.sh is the primary script for batch evaluation, this script allows the user to specify a specific seed to make it easier to investigate a particular trial. A typical workflow is to

use monkey.sh first and then use monkey-debug.sh to investigate interesting cases.

monkey.gdb : A helper script used by monkey.sh to automate GDB. Users should not need to call this script directly.

monkey.sh : Evaluation script for producing the results in Subsection 4.3 of the conference paper. Specifically, this script runs the target program several times with PUMM's protection activated, but also tries to randomly inject synthetic use-after-free vulnerabilities. The purpose of doing this is to evaluate PUMM at a larger scale than what is feasible using only real-world vulnerabilities.

procmap.sh : Helper script for extracting memory layout information from Perf recordings. Users should not need to use this script directly.

profile-name.sh : Helper script for determining the profile name for a given target program. This script is intended to help users locate a saved PUMM profile. For example, a user can provide this script with the name of a program that PUMM has already analyzed, and then they can look in PUMM's profile directory for the matching profile file.

ptdump.sh : A helper script to decode a Perf trace into a human-readable log of the Intel PT packets. Users should not need to call this script directly and is intended for debugging.

ptxed.sh : A helper script for decoding a Perf trace into the sequence of executed instructions. Users should not need to call this script directly.

trace.sh : The script for recording execution traces using Perf. See the README in the code repository for usage instructions and examples.

eval : Contains 3 additional scripts to help with evaluation. Specifically, these scripts accept a directory containing 1 or more decoded traces and calculate the number of executed instructions, total size of the traces, and estimate code coverage of the target program.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220912. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenix%20sec2023/.