SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel

Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard, *Graz University of Technology*

https://www.usenix.org/conference/usenixsecurity24/presentation/maar-slubstick

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

USENIX Security '24 Artifact Appendix: SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel

Lukas Maar
Graz University of Technology

Stefan Gast
Graz University of Technology

Martin Unterguggenberger
Graz University of Technology

Mathias Oberhuber
Graz University of Technology

Stefan Mangard
Graz University of Technology

A Artifact Appendix

A.1 Abstract

We present SLUBStick, a novel kernel exploitation technique that elevates a limited heap vulnerability to an arbitrary memory read/write primitive. SLUBStick works in several steps: Initially, it exploits a timing side channel of the allocator to reliably perform a cross-cache attack with a better than 99 % success rate on commonly used generic caches. SLUBStick then exploits code patterns prevalent in the Linux kernel to perform a cross-cache attack and turn a limited heap vulnerability into a page table manipulation, thereby granting the capability to read and write memory arbitrarily.

The artifacts demonstrate the timing side channel and end-to-end exploits, showing the versatility of SLUBStick. For both, we provide an environment of a Virtual Machine (VM) running the Linux kernel x86_64 v6.2. For the timing side channel, the evaluation presents success rates for slab pages, with and without noise. For the end-to-end exploits, our attacks exploit an artificial Double Free (DF) vulnerability to obtain an arbitrary physical read and write primitive. This primitive is then used to manipulate the `/etc/passwd` file to gain root privileges within the VM.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifacts do not perform any destructive steps as we exploit an artificial DF vulnerability introduced via a kernel module within a VM. This evaluation of the artifacts demonstrates the practicality of running SLUBStick to gain root privileges.

A.2.2 How to access

We provide the source code (github) for performing the timing side channel and the end-to-end attacks of an artificial DF vulnerability. For convenience, we provide a VM image (zenodo) with all necessary programs and scripts included.

A.2.3 Hardware dependencies

A Linux system running on the x86_64 architecture with a sound module which requires the `snd` kernel module.

A.2.4 Software dependencies

A Linux system that allows to run qemu with KVM enabled.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

1. Install qemu on your x86_64 Linux system, and make sure it is allowed to run with KVM enabled.

2. Clone our github repository (github) to the `/repo/path` directory.

3. Download the VM image (zenodo) and store it in `/repo/path/images`. This image is the default Ubuntu 22.04 image running the Linux kernel v6.2. The username to log in is `lmaar`, and the password is `asdf`. This user is in the `sudo` group and, thereby, can gain root privileges via `sudo su`.

A.3.2 Basic Test

1. Execute `make run` boots the VM, providing a terminal login prompt and an already logged in user on a graphical interface, i.e. Gnome.
2. Either login in the terminal (with username `lmaar` and password `asdf`) or open a terminal in the graphical interface.
3. Execute `uname -r` should return `6.2.0-x-generic`.
4. The directory `/home/lmaar/exploits` should include `helper.c`, `do_eval*.sh`, and `eval.py`.
5. The directory `/home/lmaar/exploits/userspace` should include `*.c`.
6. Change directory to `/home/lmaar/exploits` and execute `make init`.

A.4 Evaluation workflow

A.4.1 Major Claims

We provide artifacts verifying the following claims:

(C1): We demonstrate our proposed measurement primitives allow a timing side channel on the SLUB allocator (described in Section 4.1), leaking when a new slab is allocated. This is proven by **(E1/2/3)** with 3 different primitives.

(C2): We demonstrate that using measurement and allocation primitives, we can reliably trigger the recycling process for a targeted memory chunk from generic caches between `kmalloc-[8,4096]` (described in Section 4.1). This is proven by **(E4)** for single page slabs (i.e., `kmalloc-[8,256]`) and proven by **(E5)** for multi page slabs (i.e., `kmalloc-[512,4096]`).

(C3): We demonstrate that by using buddy allocator massaging (described in Section 4.2), we can reliably reclaim the targeted memory chunk from generic caches from `kmalloc-[8,4096]`. This is proven by **(E4/5)**.

(C4): We evaluate the reliable triggering of the recycling and reclaiming of a targeted memory chunk under idle as well as noisy conditions (described in Section 4.3 and shown in Table 1). This is proven by **(E6)** under idle and without CPU pinning and proven by **(E7)** with external noise.

(C5): We demonstrate that persistent code pattern 1/2 and temporal code pattern can be exploited to establish a memory write primitive (described in Section 5.2). This is proven by **(E8/9)** for persistent code pattern 1/2 and by **(E10)** for temporal code pattern.

(C6): We demonstrate that FUSE can be leveraged as an unprivileged user to gain control over the copying from userspace, allowing us to perform the cross-cache reuse inbetween (described in Section 5.2). This is proven by **(E8/9/10)**.

(C7): We demonstrate the exploitation of a memory write primitive to obtain an arbitrary physical read/write primitive (described in Section 6). This is proven by **(E8/9/10)**.

(C8): We evaluate the end-to-end SLUBStick attack using a synthetic DF vulnerability in generic caches between `kmalloc-[8,256]` (described in Section 7.1). This is proven by **(E8/9/10)**.

A.4.2 Experiments

Before running the experiments, please perform the set-up in A.3 and read the note in A.5.

(E1): Basic leakage 1 [30 human-seconds + 1 computer-second]:
How to: Execute `./userspace/timed_ppdev_alloc.elf`.
Cache size: Fixed with of 192 Bytes.
Results: This experiment outputs a vertical plot with the following format: `<index>: <tsc>:###`, where `index` is the allocated object's index, `tsc` the required time of the allocation, and `###` provides a visual representation of the `tsc`. Allocating a new slab will result in a significantly larger `tsc` (as shown in ⑤ from Figure 2 and described in Section 2.1) compared to that from the CPU free list (① from Figure 2). Since one slab (from the `kmalloc-192`) can include 21 objects (as shown in Table 4), a new slab will be allocated every 21th allocation. This is seen with the significantly larger `tsc` (i.e., above 2000 compared to about 1100) of every 21th index.
It is important to note that sometimes other allocations beside each 21st will cause a higher `tsc` due to noise from the system.

(E2): Basic leakage 2 [30 human-seconds + 1 computer-second]:
How to: Execute `./userspace/timed_anon_vma_name_alloc.elf <cache_size>`.
Cache sizes: 16, 32, 64, or 96 Bytes.
Results: Similar to **(E1)**, this experiment indicates larger `tsc` values when the SLUB allocator allocates a new slab. Depending on the `cache_size` (i.e., between 16 Bytes to 96 Bytes), the new slab is allocated within the `kmalloc-[16,96]`. For larger `cache_sizes`, fewer objects must be allocated to prompt the allocator to allocate a new slab. This is because fewer objects are located on one slab (as shown in Table 4). For instance, the `kmalloc-16` stores 256 objects per slab while `kmalloc-96` stores 42, indicated with significantly larger `tsc` values.

(E3): Basic leakage 3 [30 human-seconds + 1 computer-second]:

How to: Execute `./userspace/timed_msg_alloc-.elf <cache_size>`.
Cache sizes: 64, 96, 128, 192, 256, 512, 1024, 2048, or 4096 Bytes.
Results: Similar results to **(E2)**, but for the `kmalloc-cg-[64,4096]` depending on the `cache_size` (i.e., between 16 Bytes to 96 Bytes).

(E4): Single page slab reclaiming [30 human-seconds + 1 computer-second]:
How to: Execute `./userspace/eval_pud.elf <cache_size>`.
Cache sizes: 8, 16, 32, 64, 96, 128, 192, or 256 Bytes.
Results: This experiment outputs which slab page was reclaimed as a Page Upper Directory (PUD), successfully performing the cross-cache attack from the kernel heap to a PUD. If the correct slab page is reclaimed as a PUD, this experiment outputs `[+] SUCCESS`, while `[!] FAIL` indicates a failed experiment. `[!] RETRY (start not found)` indicates that the side channel did not find suitable slabs for cross-cache reuse. This output represents neither a failure nor a success, but the experiment should be repeated.

(E5): Mutli page slab reclaiming [30 human-seconds + 1 computer-second]:
How to: Execute `./userspace/eval_pmd.elf <cache_size>`.
Cache sizes: 512, 1024, 2048, or 4096 Bytes.
Results: Same results as **(E4)**.

(E6): Reclaiming on idle and without cpu pinning [30 human-seconds + 2 computer-hours]:
How to: Execute `./do_eval.sh` and then `eval.py`.
Execution: `./do_eval.sh` internally executes all `./eval_*.elf` with and without CPU pinning and outputs log files in the `./eval` directory. `./eval.py` reads all log files in `./eval`.
Results: `./eval.py` ouputs a table similar to Table 1, but without the standard deviation.

(E7): [Optional] Reclaiming with external noise [10 human-minutes + 2 computer-hours]:
How to: Execute `./do_eval_p(u,m)d_noise.sh <cache_size>` and then `eval.py`, with `pud` using objects between 8 Bytes to 256 Bytes and `pmd` using objects between 512 Bytes to 4096 Bytes
Execution: This experiment is similar to **(E6)**, but it requires rebooting the VM after each run for the `./do_eval_pmd_noise.sh` script.
Results: Same results as **(E6)**.

(E8): End-to-end exploit with persistent code pattern 1 [30 human-seconds + 1 computer-minutes]:
How to: Execute `./userspace/exploit_-signal.elf`.
Cache size: Fixed with of 8 Bytes.
Results: Tampers the `/etc/passwd` such that unprivileged users can elevate their privilege level by calling `su` without authentication.

(E9): End-to-end exploit with persistent code pattern 2 [30 human-seconds + 30 computer-seconds]:
How to: Execute `./userspace/exploit_snd.elf <cache_size>`.
Cache sizes: 16, 32, 64, 96, 128, 192, or 256 Bytes.
Results: Same results as **(E8)**.

(E10): End-to-end exploit with temporal code pattern [30 human-seconds + 30 computer-seconds]:
How to: Execute `./userspace/exploit_key.elf <cache_size>`
Cache sizes: 16, 64, 96, or 128 Bytes.
Results: Same results as **(E8)**.

A.5 Notes on Reusability

For this artifact evaluation, we evaluate generic caches that allocate memory in chunks of 8, 16, 32, 64, 96, 128, 192, 256, 512, 1024, 2048, and 4096 Bytes. Some experiments only work on certain generic cache sizes, while others work generically on more.

We want to note that the success rate of the end-to-end exploit **(E8-10)** varies depending on the cache size. Moreover, since these exploits corrupt the memory, (successfully) triggering them multiple times may cause the VM to crash.

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/.