USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks

Lukas Maar, Lukas Giner, Daniel Gruss, and
Stefan Mangard, *Graz University of Technology*

https://www.usenix.org/conference/usenixsecurity25/presentation/maar-kernel

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

USENIX Security '25 Artifact Appendix: When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks

Lukas Maar
Graz University of Technology

Lukas Giner
Graz University of Technology

Daniel Gruss
Graz University of Technology

Stefan Mangard
Graz University of Technology

A Artifact Appendix

A.1 Abstract

This paper shows how side-channel leakage in kernel defenses can be exploited to leak the locations of security-critical kernel objects, enabling reliable and stable attacks on the Linux kernel. By systematically analyzing 127 defenses, we show that enabling any of three specific defenses – strict memory permissions, kernel heap virtualization, or stack virtualization – exposes fine-grained TLB contention patterns. These patterns are then combined with kernel allocator massaging to perform location disclosure attacks, revealing the locations of kernel heap objects, page tables, and stacks.

The artifacts demonstrate the timing side-channel attack and the exploit techniques. For both, we provide a kernel module and programs to perform the experiments.

1. For the timing side channel, we leak the location of kernel heap objects (i.e., `pipe_buffer`, `msg_msg`, `cred`, `file`, and `seq_file`), page tables (all levels), and the kernel stack. While our side channel works on all Intel generations between the 8th and 14th, we recommend evaluating on Intel 13th generation, as we have mainly evaluated on this one. While our side channel works on Linux kernels between v5.15 and v6.8, we recommend evaluating on the Ubuntu generic kernel v6.8.
2. For the exploit techniques, we perform privilege escalation using the 3 techniques. Inherent to kernel exploitation, we tailor these techniques to the specific Ubuntu generic kernel v6.8.0-38, the required version to evaluate these techniques.

A.2 Description & Requirements

A.2.1 Security, Privacy, and Ethical Concerns

For evaluating the timing side channel, the experiments can be used in kernel exploitation of memory-corruption attacks, as they allow the location of kernel objects be leaked on Intel CPUs. This raises potential ethical concerns.

For evaluating the exploit techniques, the artifacts might result in destructive steps. While we introduce an exploit primitive via a kernel module and do not provide methods to compromise systems in the wild, the experiment using this primitive may cause system crashes. However, during our evaluation, we have not encountered a single system crash.

A.2.2 How to Access

We provide the source code (github and zenodo) for performing the timing side channel.

A.2.3 Hardware Dependencies

A Linux system running on an Intel CPU between 8th and 14th generation. However, we recommend running the evaluation on a 13th generation Intel, as this is what we have mainly evaluated on. We have observed a trend that newer Intel CPUs and better cooling tend to give more stable results.

A.2.4 Software Dependencies

While our disclosure attacks should generically work on Linux kernels, our experiments are tailored to Ubuntu Linux kernels between v5.15 to v6.8. As reference, we have mainly evaluated on the generic Ubuntu kernel v6.8.0-38 (and the kernel with `CONFIG_SLAB_VIRTUAL` of v6.6[1]).

One part of the artifact evaluation is to insert a kernel module that requires *root privileges*. This module is required to obtain the ground truth of the kernel object's location as well as for providing the exploit primitive for the exploit techniques. We tested our kernel module on Ubuntu Linux downstream kernels v5.15, v6.5, and v6.8 and kernel v6.6. For other kernels that have different config files (or other downstream changes) our implemented module may not do what we intended. Specifically, in order to obtain the location of kernel objects, we redefined and reimplemented structures

[1] https://lore.kernel.org/all/CAHKB1wLetbLZjhg1UVhA1QwZHo226BRL=Khm962JEfh0F+CVbQ@mail.gmail.com/T/

and functions. We did this because some functions used to access kernel data structures are implemented as inline functions, e.g., `ipc_obtain_object_check`, which prevented us from calling these functions directly. Another reason is that some structs, e.g., `msg_queue`, are defined in c files, which also prevents us from using these struct definitions.

While this artifact evaluation include experiments exploiting leakage for all defenses, we only recommend to reproduce experiments exploiting leakage from **D1** and **D3**. This is because reproducing leakage from **D2** requires a Linux kernel compiled with `CONFIG_SLAB_VIRTUAL`, i.e., the intended v6.6 used by Google's KernelCTF. We encountered driver crashes during boot due to incompatibilities, requiring additional engineering effort. However, all of our heap location leakage attacks should work directly by exploiting **D2** when swapping the base address from the DPM to the virtual heap.

Due to the nature of kernel exploitation in general, our exploit techniques depend on the kernel version. Therefore, we only provide the end-to-end attack for the exact Ubuntu Linux kernel v6.8.0-38 (and the kernel with `CONFIG_SLAB_VIRTUAL` of v6.6). The exact version is needed, e.g., for the control-flow hijacking attack, as the ROP chain varies between versions. Similarly, the other two exploit techniques require internal version-dependent kernel information. Curically, all information can be obtained as an unprivileged user but requires engineering effort.

A.3 Set-Up

A.3.1 Installation

The installation works as following:

1. Clone our github repository (github) to `/repo/path` directory.
2. Change directory to `/repo/path`.
3. Select in `./lkm.c` either `V5_15`, `V6_5`, `V6_6`, or `V6_8`, depending on your running Ubuntu Linux kernel version.
4. Execute `make init` to build the kernel module and all experiments and insert the kernel module.

A.3.2 Basic Test

Before starting the experiments, determine the TLB hit thresholds on your CPU as follows. It is important to ensure that the background noise is as low as possible before starting this basic experiment, as described in A.5.

1. Change directory to `/repo/path/generic`.
2. Execute `./threshold_detection.elf` prints `[+] detected thresholds: <THRESHOLD> <THRESHOLD2>`, where `THRESHOLD` is the threshold for capturing 97 % of all hit timings of mapped addresses and `THRESHOLD2` is the threshold for the minimum timing of unmapped addresses.
3. Repeat this a few times and write the most consistent result to `THRESHOLD` and `THRESHOLD2` in `/repo/path/include/tlb_flush.h` and recompile with `make build` in `/repo/path`.

A.4 Evaluation Workflow

A.4.1 Major Claims

We provide artifacts verifying the following claims:

(C1): We demonstrate to leak the base address of the DPM, i.e., `page_offset_base`. This is proven by **(E1)** described in Section 6.2 **Leaking Coarse-Grained Kernel Section**.

(C2): We demonstrate to leak the base address of the memory location used by `vmalloc`. This is proven by **(E2)** described in Section 6.2 **Leaking Coarse-Grained Kernel Section**.

(C3): We demonstrate to leak the base address of the virtual memory mapping `vmemmap`. This is proven by **(E3)** described in Section 6.2 **Leaking Coarse-Grained Kernel Section**.

(C4): We demonstrate that exploiting **D1** allows to leak the page-aligned location of `msg_msg`. This is proven by **(E4)** described in Section 5.1 and Section 6.2 **Leaking Fine-Grained Locations** and shown in Table 1.

(C5): We demonstrate that exploiting **D1** allows to leak the page-aligned location of `file`. This is proven by **(E5)** described in Section 5.1 and Section 6.2 **Leaking Fine-Grained Locations** and shown in Table 1.

(C6): We demonstrate that exploiting **D1** allows to leak the page-aligned location of `seq_file`. This is proven by **(E6)** described in Section 5.1 and Section 6.2 **Leaking Fine-Grained Locations** and shown in Table 1.

(C7): We demonstrate that exploiting **D1** allows to leak the page-aligned location of `pipe_buffer`. This is proven by **(E7)** described in Section 5.1 and Section 6.2 **Leaking Fine-Grained Locations** and shown in Table 1.

(C8): We demonstrate that exploiting **D1** allows to leak the locations of page-tables, i.e., Page Table (PT), Page Middle Directory (PMD), and Page Upper Directory (PUD). This is proven by **(E8)** described in Appendix B and Section 6.2 **Leaking Fine-Grained Locations** and shown in Table 1.

(C9): We demonstrate that exploiting **D3** allows to leak the location of the kernel stack. This is proven by **(E9)** described in Section 5.3 and Section 6.2 **Leaking Fine-Grained Locations** and shown in Table 1.

(C10): We demonstrate high reliability of our location disclosure attacks. This is proven by **(E10)** shown in Table 1.

(C11): We demonstrate the **Unlink Primitive** exploit technique. This is proven by **(E11)** described in Section 7.1.

(C12): We demonstrate the **Use-After-Free & Out-Of-Bounds Write** exploit technique. This is proven by **(E12)** described in Section 7.1.

(C13): We demonstrate the **Constrained Write Primitive**

exploit technique. This is proven by **(E13)** described in Section 7.1.

(C14): We demonstrate high reliability of our exploit techniques. This is proven by **(E14)**.

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 DPM location leakage [10 human-seconds + 1 computer-second]:
How to: Execute `./generic/dpm_leak.elf`.
Results: This experiment outputs the base address of the DPM.

(E2): Basic `vmalloc` memory location leakage [10 human-seconds + 1 computer-second]:
How to: Execute `./generic/vmalloc_leak.elf`.
Results: This experiment outputs the base address of the virtual memory section used for `vmalloc`.

(E3): Basic `vmemmap` memory location leakage [10 human-seconds + 1 computer-second]:
How to: Execute `./generic/vmemmap_leak.elf`.
Results: This experiment outputs the base address of the virtual memory mapping `vmemmap`.

(E4): `msg_msg` location leakage [10 human-seconds + 10 computer-seconds]:
How to: Execute `./heap/msg_msg_leak.elf`.
Results: This experiment outputs the page-aligned `msg_msg` object location.

(E5): `file` location leakage [10 human-seconds + 10 computer-seconds]:
How to: Execute `./heap/file_leak.elf`.
Results: This experiment outputs the page-aligned `file` object location.

(E6): `seq_file` location leakage [10 human-seconds + 10 computer-seconds]:
How to: Execute `./heap/seq_file_leak.elf`.
Results: This experiment outputs the page-aligned `seq_file` object location.

(E7): `pipe_buffer` location leakage [10 human-seconds + 10 computer-seconds]:
How to: Execute `./heap/pipe_buffer_leak.elf`.
Results: This experiment outputs the page-aligned `pipe_buffer` object location.

(E8): Page-table location leakage [10 human-seconds + 20 computer-seconds]:
How to: Execute `./page-table/pt_leak.elf`, `./page-table/pmd_leak.elf`, or `./page-table/pud_leak.elf`.
Results: This experiment outputs the respective location of PT, PMD, and PUD.

(E9): Kernel stack location leakage [10 human-seconds + 2 computer-seconds]:
How to: Execute `./stack/stack_leak.elf`.
Results: This experiment outputs the current kernel stack location.

(E10): Reliable location disclosure attacks [5 human-minute + 2 computer-hours]:
How to: Execute `./eval.sh` (in `heap`, `page-table` and `stack`) and then `./print.py`.
Description: The `./eval.sh` scripts performs between 20 to 100 execution of (E4-9) depending on the experiment, while `./print.py` prints a table which should closely resemble Table 1. As described in A.5, background activity should be minimized in this evaluation.
Results: This experiment outputs the content of Table 1 for **D1** and **D3**.

(E11): Unlink primitive [10 human-seconds + 1 computer-seconds]:
How to: Execute `./attacks/pipe_unlink.elf`.
Limitation: Works with Ubuntu kernel v6.8.0-38. Other versions will most likely lead to a program crash.
Results: Privilege escalation.

(E12): Use-After-Free & Out-Of-Bounds write [10 human-seconds + 1 computer-seconds]:
How to: Execute `./attacks/dirty_page.elf` or `./attacks/advanced_slubstick.elf`.
Limitation: Works with Ubuntu kernel v6.8.0-38 or the v6.6 intended to be used with `CONFIG_SLAB_VIRTUAL`. Other versions will most likely lead to a program crash.
Results: Privilege escalation.

(E13): Constrained write [10 human-seconds + 3 computer-seconds]:
How to: Execute `./attacks/stack_attack.elf`.
Limitation: Works with Ubuntu kernel v6.8.0-38. Other versions will most likely lead to a program crash.
Results: Privilege escalation.

(E14): Reliable exploit techniques [5 human-minutes + 10 computer-minutes]:
How to: Execute `./eval.sh` and then `./print.py`, both in `attacks`.
Results: This experiment shows the success rate of the 3 exploit techniques.

A.5 Notes on Reusability

As described in Section 6.2 **Stress**, the most dominant noise source is CPU frequency fluctuation. Hence, perform all experiments with as little background activity as possible to reproduce the paper's results. We even suggest to perform the experiments on an idle system with no other activity.

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/usenixsec2025/.