Cache Me If You Can: A reimplementation of SLUBStick attack for Linux 5.14

L. Sinibaldi, O. Quin

September 2, 2026

Contents

1 Project data **1**

2 Project description **2**
2.1 Design and implementation 2

3 Project outcomes **3**
3.1 Concrete outcomes 3
3.2 Learning outcomes 4
3.3 Existing knowledge 4
3.4 Problems encountered 5

4 Honor Pledge **5**

NOTE: IT IS MANDATORY TO FILL IN ALL THE SECTIONS AND SUBSECTIONS IN THIS TEMPLATE

1 Project data

- Project supervisor: Claudio Migliorelli

	Last and first name	Person code	Email address
•	Sinibaldi Leonardo	10903606	leonardo.sinibaldi@mail.polimi.it
	Quin Oliver	10809040	oliver.quin@mail.polimi.it

- Task division between group's members:
  - Leonardo worked on implementing the timing side channel to control slub allocator
  - Leonardo worked on implementing the infrastructure for the arbitrary memory write primitive, in particular referred to the corruption of `/etc/passwd`
  - Leonardo worked on implementing the CVE on `route4_change`
  - Leonardo worked on implementing the memory write primitive trigger using `replace_user_tlv` function in sound subsystem
  - Oliver worked on the page recycling mechanism and synchronization between threads using pipes for communications
  - Oliver worked on porting the primitives from a custom module to real kernel primitives
  - Oliver worked on FUSE technique for expanding the recycling time window
- github.com/LSinus/CacheMeIfYouCan

2 Project description

2 pages max please

- What is your project about?
- Why it is important for the AOS course?

Cache Me If You Can is an exploit based on the page recycling technique, enhanced with a timing side-channel as proposed in SLUBStick. In this project, we primarily explored the SLUB allocator, alongside other subsystems such as the traffic control, ALSA and the Distributed Lock Manager.

This project is tightly related to the topics covered in the Advanced Operating Systems (AOS) course. The entire page recycling mechanism relies on understanding how the SLUB allocator handles CPU caches, node caches, and slab freeze status, as well as how freeing all objects within a slab returns the underlying page to the buddy allocator rather than keeping it for some cache.

Since modern kernels use cache isolation to improve security, the original authors leveraged a cross-cache attack. This allows a benign vulnerability in a less critical cache, like the generic `kmalloc-128`, to be exploited on highly sensitive caches, such as the page table caches. Thus, recycling a vulnerable object from the generic cache to the protected cache effectively bypasses the isolation security measure.

Understanding the side channel requires knowing the various paths an object allocation can take, ranging from the fastest path via the `kmem_cache_cpu->freelist` to the slowest, which involves requesting a new page from the buddy allocator. Exploiting this timing side channel was vital to significantly improve the reliability of the recycling process.

The side-channel is the core contribution of the original paper; considering the high level of concurrency and noise within a running kernel, having a timing side-channel that reveals exactly when a particular allocation happens is the critical enabler. This mechanism pushed the exploit's success rate to over 99% for all generic caches up to 256 bytes and with lower scores for bigger caches.

Furthermore, we explored the memory management subsystem and demonstrated how, once a vulnerable object is recycled as a Page Upper Directory (PUD), we can bypass KASLR protection. By linearly scanning the kernel address space, we gain the ability to alter arbitrary data; in our specific case, we successfully overwrote `/etc/passwd` file to trick the kernel into thinking that the `root` user had no password.

2.1 Design and implementation

To construct the exploit, we required several foundational building blocks. The first component we implemented was the measurement primitive (MP). For an allocation to serve as an effective MP, it must perform minimal work between allocation and deallocation; for this purpose, `__do_sys_add_key` works particularly well. First, it allows us to allocate objects in any general-purpose slab we target. Second, if the provided key is invalid, execution immediately jumps to the `ERR` label, cleaning up the newly allocated object. Timing this system call provides an excellent measurement primitive to infer the state of the target cache.

We also needed to implement two additional primitives: an Allocation Primitive (AP) and a Memory Write Primitive (MWP). Because these depend on the target cache, which in our case was `kmalloc-128`, we leveraged `dlm_user_proc` for the AP and `key_ctl_key_verify` for the MWP. For the Double-Free (DF) primitive, we relied on CVE-2022-2588, first initializing the vulnerable `route4` configuration and preparing the corresponding `route4_filter`, which is later used to trigger the double free.

With our primitives successfully implemented, we proceeded to develop the page recycle and reclaim execution flow. The original paper proposed two approaches for this step: one utilizing a persistent code pattern, where a kernel object remains allocated and accessible across multiple operations, allowing the attacker to trigger a write to the same object after the underlying slab page has been recycled and reclaimed; and another leveraging the Filesystem in Userspace (FUSE) to artificially extend the time window between allocation and deallocation. We opted for the latter approach.

The FUSE-based exploit utilizes three separate processes synchronized via pipes. To begin, the main thread, hereafter referred to as processeA for clarity, , spawns the FUSE processthread and enters a sleep state, allowing FUSE

sufficient time to initialize and mount the FS. Following this, processA prepares the environment by creating a namespace within the audio subsystem to facilitate the Double Free (DF) vulnerability. Finally, processA spawns a secondary process, designated as processB.

ProcessA begins by using the AP and MP to obtain a fresh slab, allocating padding objects and the vulnerable object to fill the victim cache and create the dangling pointer. ProcessA then signals processB via pipes to proceed and enters a busy wait. ProcessB triggers the MWP through a syscall that performs a `copy_from_user()` from a memory region backed by FUSE. When the kernel accesses this region, a page fault is generated, and execution is redirected to the FUSE process, which is configured to block until it receives a signal through a pipe. This intentional delay provides processA with the necessary execution window to manipulate the partial lists, trigger the DF to return the page to the buddy allocator, and reallocate the victim slab as a Page Table Entry (PTE) by performing multiple `mmap` calls from user space. Finally, processA signals FUSE to resume; the pending `copy_from_user()` in processB then completes, causing the MWP to write attacker-controlled data directly over the newly allocated PTE. The goal of this write is to change the Page Frame Number (PFN) as well as the huge page bit of the allocated PTE, allowing the exploit to access a wider physical range from a virtual address.

Once the page-table page has been successfully reclaimed, a 4 KB page mapped in the process address space is turned into a 1 GB read/write accessible window over physical memory. To convert this window into an arbitrary MWP, we first allocate a 512 MB virtual memory region and populate each 4 KB page with a unique marker composed of the string `MAGIC` followed by its page index. We then scan the physical 1 GB window for these markers. Since this window provides a linear mapping of physical memory, the offset at which a marker is found directly identifies the physical address of the corresponding page.

Once a physical offset of a marked page is found, the full 1 GB window is scanned again for the PTE that is mapping the marked page. Because a 1 GB window does not guarantee finding the target PTE, allocating a large number of pages (512 MB) ensures the process is reliable.

When such an entry is found, we modify it through the 1 GB window and verify that it corresponds to the active userspace PTE by observing the effect on the associated virtual page. This gives us direct control over a valid PTE and therefore allows us to repeatedly change the physical frame mapped by a userspace virtual address, making it possible to arbitrarily read and modify the entire physical memory.

Finally, we open `/etc/passwd` to force its contents into the page cache and use the controlled PTE to search for its physical memory location. If the page already lies within the initial 1 GB window, it can be located directly by searching for the `root:x:0:0:` prefix. Otherwise, we iteratively modify the PFN of the controlled PTE to map different physical pages into the same virtual address, effectively scanning physical memory until the same signature is found. Once the page containing `/etc/passwd` is identified, we directly modify its page-cache contents by removing the `x` character from the root entry. As a result, until the page cache content is flushed, subsequent accesses to `/etc/passwd` observe the modified page through the normal page-table translation, causing the system to treat the `root` account as having no password.

3 Project outcomes

GitHub: LSinus/CacheMeIfYouCan

3.1 Concrete outcomes

The repository includes a video demonstration of the exploit in action. The video shows a Linux virtual machine running within QEMU as an unprivileged user, alongside a Telnet login screen. Initially, Telnet prompts for the root password; however, following the execution of the exploit, it grants root access without any authentication.

Additionally, the repository contains a setup script that downloads the specific Linux kernel tarball used for the exploit and prepares the filesystem. Once the kernel and the required modules are compiled, the virtual machine can be started by executing `run.sh` inside the `vmachine` directory. By following the instructions in the `README`, the exploit can be reliably reproduced locally.

For those interested in observing the allocator's behavior during runtime, a GDB session can be attached using `target remote :1234`. Our custom module includes a dedicated kernel breakpoint; by loading the module's symbols,

one can break at `rdd_bp` and inspect the cache state using the `slab info kmalloc-128` command at various stages of the exploit. We also provide custom GDB scripts to facilitate low-level inspection: `walk_freelist` traverses the freelist of a selected slab, allowing its free objects and their ordering to be inspected, while `walk_pte` performs a page-table walk for a given virtual address, reporting the locations and contents of the entries at each level of the page-table hierarchy involved in its translation. Finally, the Linux `buddy` GDB script can be used to inspect the current state of the buddy allocator, making it possible to observe when the recycled slab page is returned to, and subsequently reallocated from, the buddy system.

3.2 Learning outcomes

Throughout the development of this project, we explored several topics related to operating systems and system programming. We began by setting up the virtual machine environment required to reproduce and analyze the exploit, while also developing custom GDB scripts to inspect the internal state of the kernel during execution. This provided us with the necessary tooling to observe allocator behavior and debug the different stages of the attack.

A significant part of the project focused on understanding and interacting with the SLUB allocator, particularly the mechanisms involved in slab allocation, deallocation, and cross-cache attacks. By manipulating the allocator state and observing how objects and pages move between different caches and the buddy allocator, we gained a practical understanding of concepts that are otherwise difficult to observe directly.

The project also allowed us to experience the complete development lifecycle of a kernel exploit. We initially worked with a synthetic kernel module designed to reproduce the core primitives and mechanisms in a controlled environment. Once these concepts were understood and reliably implemented, we progressively replaced the synthetic components with real kernel interfaces and vulnerabilities, eventually adapting the exploit to operate against an actual CVE.

Similarly, we learned how to work with Linux namespaces to reproduce and trigger vulnerabilities without requiring the exploit process to run directly as root, while still obtaining the privileges needed. We also gained practical experience in coordinating multiple processes through pipes, using them to synchronize the different stages of the exploit and control their execution order. Finally, the project required us to become familiar with the Linux kernel build system, learning how to read and modify its configuration, compile the kernel and its modules, and integrate the components required by the exploit environment.

- Leonardo learned how to set up a Linux virtual machine suitable for exploit development and testing.
- Oliver learned how to use pipes to synchronize multiple processes and how to debug concurrent execution.
- Leonardo learned how to manipulate Page Table Entries (PTEs) to overwrite the `/etc/passwd` file.
- Oliver learned how the SLUB allocator works and how to manipulate its state through slab-massaging techniques.
- Leonardo learned how to extend GDB by developing custom Python scripts.
- Oliver learned how to use GDB to inspect and debug the behavior of the SLUB allocator.
- Leonardo learned how to to interact with the `route4` subsystem to implement the exploit
- Oliver leanrd how to interact with FUSE to extend the recycling timing window
- Both Oliver and Leonardo learned how to use standard Unix utilities when working with the Linux kernel source tree, including searching for definitions, navigating the codebase, and identifying the resources and components required during exploit development.

3.3 Existing knowledge

The Computer Security and Offensive and Defensive Cyber Security courses also provided useful background for understanding the development of the exploit, the tools of the craft, and the general methodology used to turn a software flaw into a working exploit.

A possible improvement to the course could be the introduction of additional homework assignments aimed at gradually bridging the gap between general C programming and kernel development. While the laboratory sessions are valuable, they can initially feel overwhelming for students with limited experience in kernel programming. A more progressive set of exercises could help students become familiar with kernel APIs, debugging tools, and development workflows before approaching the more complex laboratory tasks.

3.4 Problems encountered

During this project, we encountered a wide range of both new and common challenges. In particular, we had to learn how userspace actions can interact with kernel space and how different kernel interfaces can be leveraged to trigger and control the behavior required by the exploit. As the project evolved, each new component often required enabling additional kernel subsystems, forcing us to become familiar with the kernel configuration and build process and to repeatedly rebuild the kernel with the relevant features enabled.

Furthermore, this was our first experience working with research material at this level of technical depth. We therefore had to learn how to approach and study research papers, understand the mechanisms and assumptions behind the proposed techniques, and translate those ideas into a concrete implementation. This required not only reproducing the concepts described in the original work, but also adapting them to our kernel version, environment, and target vulnerability.

4 Honor Pledge

(**This part cannot be modified and it is mandatory to sign it**)

I/We pledge that this work was fully and wholly completed within the criteria established for academic integrity by Politecnico di Milano (Code of Ethics and Conduct) and represents my/our original production, unless otherwise cited.

I/We also understand that this project, if successfully graded, will fulfill part B requirement of the Advanced Operating System course and that it will be considered valid up until the AOS exam of Sept. 2022.

Group Students' signatures