Oops! It's a kernel stack use-after-free: Exploiting NVIDIA's GPU Linux drivers

Robin Bastide

This article details two bugs discovered in the NVIDIA Linux Open GPU Kernel Modules and demonstrates how they can be exploited. The bugs can be triggered by an attacker controlling a local unprivileged process. Their security implications were confirmed via a proof of concept that achieves kernel read and write primitives.

The NVIDIA Open source driver

Back in 2022, NVIDIA started distributing the Linux Open GPU Kernel Modules. Since 2024, using these modules is officially "the right move" for both consumer and server hardware. The driver provides multiple kernel modules, the bugs being found in `nvidia.ko` and `nvidia-uvm.ko`. They expose ioctls on device files, most of them being accessible to unprivileged users. These ioctls are meant to be used by NVIDIA's proprietary userland binaries and libraries. However, using the header files provided in the kernel modules repository as a basis, it's possible to make direct ioctl calls.

While manually probing the attack surface related to memory allocation and management we found two vulnerabilities. They were reported to NVIDIA and the vendor issued fixes in their NVIDIA GPU Display Drivers update of October 2025

Bug #1: Kernel null-pointer dereference in `nvidia-uvm` module (CVE-2025-23300)

The `UVM_MAP_EXTERNAL_ALLOCATION` ioctl of the `nvidia-uvm` module allows mapping memory allocated from the main `nvidia` module into the Unified Virtual Memory framework. This includes memory allocations of type `NV01_MEMORY_DEVICELESS` which are not associated with any device and therefore have the `pGpu` field of their corresponding `MEMORY_DESCRIPTOR` structure set to null. The ioctl call leads to an unchecked use of this field, resulting in a kernel null-pointer dereference. An example stack trace is provided below:

```
// linux 6.11.0-24 + nvidia 570.86.15 from Ubuntu Noble

osIovaMap+0x11e/0x630 [nvidia]
iovaspaceAcquireMapping_IMPL+0x232/0x470 [nvidia]
memdescMapIommu+0x90/0x300 [nvidia]
```

```
dupMemory+0x2d9/0x830 [nvidia]
nvUvmInterfaceDupMemory+0x44/0xe0 [nvidia]
uvm_map_external_allocation_on_gpu+0x298/0x500 [nvidia_uvm]
uvm_api_map_external_allocation+0x5dd/0x860 [nvidia_uvm]
uvm_ioctl+0x1aad/0x1e70 [nvidia_uvm]
uvm_unlocked_ioctl_entry.part.0+0x7b/0xf0 [nvidia_uvm]
uvm_unlocked_ioctl_entry+0x6a/0x90 [nvidia_uvm]
__x64_sys_ioctl+0xa3/0xf0
x64_sys_call+0x11ad/0x25f0
do_syscall_64+0x7e/0x170
```

🛠️✅ NVIDIA Fix

A new check was added to the function `dupMemory` so that operations that require valid GPU contexts are skipped for deviceless memory.

Bug #2: Kernel use-after-free in `threadStateInit()` and `threadStateFree()` in `nvidia` module (CVE-2025-23280)

The `threadStateInit()` and `threadStateFree()` functions are used in multiple locations of the `open-gpu-kernel-modules` codebase. They are always used as a pair to encapsulate specific operations, as seen in the following example:

```
// src/nvidia/src/kernel/rmapi/mapping.c (line 433)

NV_STATUS
rmapiMapWithSecInfoTls
(
    RM_API             *pRmApi,
    NvHandle            hClient,
    NvHandle            hDevice,
    NvHandle            hMemCtx,
    NvHandle            hMemory,
    NvU64               offset,
    NvU64               length,
    NvU32               flags,
    NvU64              *pDmaOffset,
    API_SECURITY_INFO  *pSecInfo
)
{
    THREAD_STATE_NODE threadState;
    NV_STATUS         status;
```

```
    threadStateInit(&threadState, THREAD_STATE_FLAGS_NONE);

    status = rmapiMapWithSecInfo(pRmApi, hClient, hDevice, hMemCtx,
hMemory, offset,
                                 length, flags, pDmaOffset, pSecInfo);

    threadStateFree(&threadState, THREAD_STATE_FLAGS_NONE);

    return status;
}
```

The `threadState` structure will be inserted into a global red-black tree (`threadStateDatabase.dbRoot`) during `threadStateInit()` and removed during `threadStateFree()`. The fact that this structure is always stack-allocated is dangerous if a kernel oops occurs between the two function calls. The oops will lead to the kernel stack for this task being freed on modern Linux kernels, which use virtual stacks allocated through `vmalloc`. As a result, an invalid pointer to the now freed stack would remain in the global tree structure. This is exactly what happens when bug #1 is triggered: `threadStateInit()` is called during `dupMemory()` (in `src/nvidia/src/kernel/rmapi/nv_gpu_ops.c`) and the null-pointer dereference happens before the call to `threadStateFree()`. The following stack trace shows the use-after-free being triggered by a call to `open` on `/dev/nvidia0` after the oops caused by bug #1:

```
// linux 6.11.0-24 + nvidia 570.86.15 from Ubuntu Noble

_mapInsertBase+0x3c/0x320 [nvidia]
threadStateInit+0xd5/0x1b0 [nvidia]
rm_is_device_sequestered+0x28/0x60 [nvidia]
nv_open_device+0x2ef/0x9e0 [nvidia]
nvidia_open+0x22a/0x4b0 [nvidia]
chrdev_open+0xd2/0x250
do_dentry_open+0x218/0x4c0
vfs_open+0x30/0x100
do_open+0x2ba/0x440
path_openat+0x132/0x2c0
do_filp_open+0xc0/0x170
do_sys_openat2+0xb3/0xe0
__x64_sys_openat+0x55/0xa0
x64_sys_call+0x230a/0x25f0
do_syscall_64+0x7e/0x170
```

> 🛠️✅ NVIDIA Fix
>
> The heap based `threadStateAlloc` function was added as a "new UAF-safe API". However, it seems it is currently used as a replacement for the stack based `threadStateInit` only in the `dupMemory` function. This has not been tested, but, other functions still using `threadStateInit` may continue to be vulnerable to a UAF in the case of a oops.

Exploitation

Proof of concept exploitation was carried out in the following environment:

ThinkPad P14s Gen 3 (Intel) with NVIDIA T550 Laptop GPU

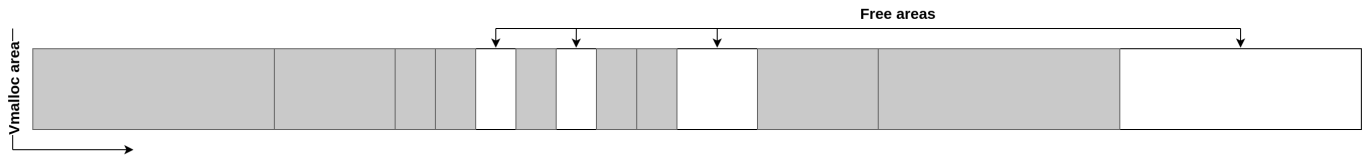
Ubuntu Noble with the following packages:
linux-image-6.11.0-24-generic (6.11.0-24.24~24.04.1 amd64)

nvidia-driver-570-server-open (570.86.15-0ubuntu0.24.04.4 amd64)

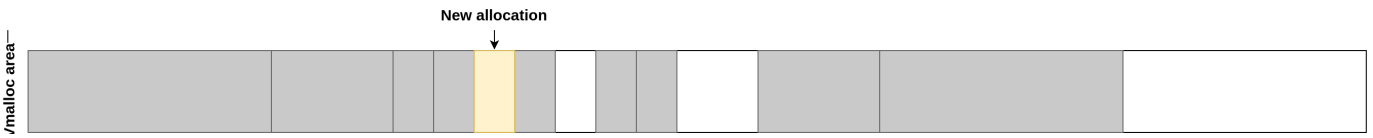
Since bug #1 is only used to trigger bug #2, we will focus on the latter. This bug is quite unusual since the UAF address is part of a kernel stack, and as such it belongs to a `vmalloc` area. Most resources available on UAF exploitation are related to `kmalloc` as it's used way more broadly for kernel allocations. The only reference for exploitation related to `vmalloc` seems to be "An iOS hacker tries Android" from Brandon Azad. However, things changed since then, for example the introduction of `random_kstack_offset`. This feature introduces a randomly generated stack offset at each syscall entry, effectively cancelling its mostly deterministic layout. By randomising the position of key stack values, it makes exploitation more difficult.

Vmalloc

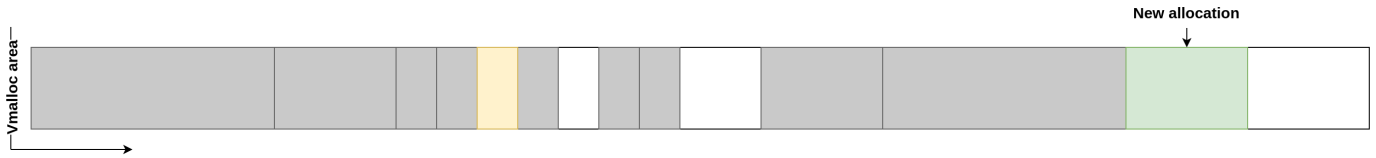
`vmalloc` is a kernel function for allocating virtually contiguous memory with a page granularity. It's notably used for allocating kernel stacks, as well as other large kernel allocations. On a running system, the allocations can be inspected using `/proc/vmallocinfo`. This section will discuss the behavior of the allocator, focusing on address space management, without addressing how backing pages are selected. Here is a very simplified representation of an area managed by `vmalloc`:



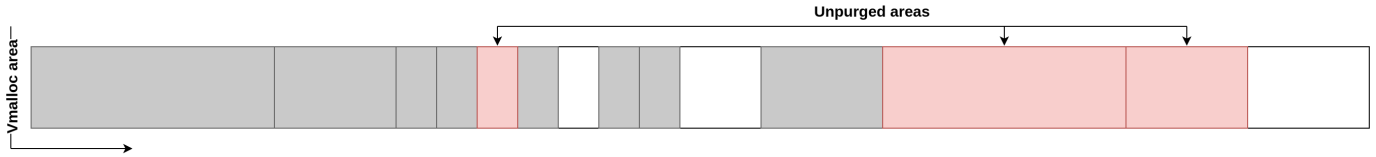
When a new allocation is made, it's placed in the first free area that can accommodate its size. Here is an example for a small allocation that takes the first empty slot:



Here is an example for a bigger allocation that didn't fit in the first available slot and so is being allocated further away:



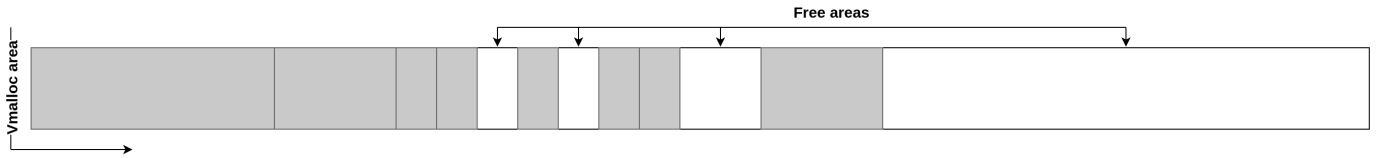
When allocations are released, they are not immediately freed but instead marked as unpurged. While they are not used by the kernel anymore, they still live in the `vmalloc` area and the address cannot be reused directly. Here is an example if we free three of the allocations:



To be effectively freed, the unpurged allocations must be purged. This is done when the number of pages contained in the unpurged allocations crosses the value returned by `lazy_max_pages`, which can easily be computed from userland and is defined as follows:

```
// linux/mm/vmalloc.c

static unsigned long lazy_max_pages(void)
{
    unsigned int log;

    log = fls(num_online_cpus());

    return log * (32UL * 1024 * 1024 / PAGE_SIZE);
}
```

After the purge, all released areas are typically ready to be used again for allocations:



However, due to recent optimisations, the kernel will now add freed allocations back into size-based pools. While they are in these pools, they will be reused in priority for allocations of the same size and the corresponding areas cannot be used for allocations of other sizes. This is a bit annoying in the context of the exploitation of a UAF, but the pools have a "decay" feature where ~25% of their contents will be released during a purge. By triggering a lot of purges instead of one,

we can completely empty out the pools and get a similar result to the old behavior.

Shaping primitives

To act on the `vmalloc` area from an unprivileged process we will use the three following primitives.

Forking

As previously mentioned, kernel stacks are allocated in the `vmalloc` area. As each userland process has its own dedicated kernel thread stack, forking will lead to a new 0x5000 bytes allocation. This corresponds to four pages for the stack itself and one guard page. Freed kernel stacks are cached to be possibly reused later without the need for new allocations. However, when a stack is released, the operation is usually delayed meaning that if we write very aggressive code like this:

```
while (1) {
    if (fork() == 0) {
        exit(0);
    }
}
```

It will lead to the stack cache not being used properly, triggering numerous allocations and deallocations, ultimately leading to a lot of unpurged areas.

Video4linux2 buffers

The v4l2 (video4linux2) framework is used for interacting with video devices from userland. It has nothing to do with the NVIDIA driver but it can provide some powerful `vmalloc` capabilities. Indeed, it has a `vmalloc` backend for allocating buffers shared with the user (`drivers/media/common/videobuf2/videobuf2-vmalloc.c`). The use of this backend is not systematic but seems to be common for internal and external USB-based webcams. The target system being a laptop, it's of course fit with one such device. However, some systems may restrict the use of video devices to the `video` group.

By opening a video device using the `vmalloc` backend we get access to the following capabilities:

Allocate between 1 and 16 buffers at once

Control the size by asking for different resolutions

`mmap` the buffers in userland while they are also mapped in kernel

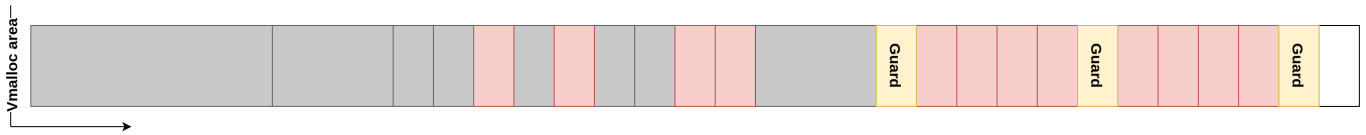
Only one set of buffers can be allocated per video device. However, the `mmap` capability is extremely powerful and the fact that we can allocate large buffers is also very useful to generate a lot of unpurged pages to trigger purges.

Side effect purge

We know that we can trigger purges by allocating and freeing a large number of buffers using either forking or v4l2 buffers. Still, it's not possible to know precisely when the purge will happen. However, exceeding `lazy_max_pages` unpurged pages is in fact not the only way to cause a purge. And, by sheer chance, the allocation of a deviceless memory inside the NVIDIA driver (i.e. the type of memory used to trigger bug #1) will cause `nv_alloc_contig_pages()` to be called with the `NV_MEMORY_UNCACHED` flag. This will cause an attribute change using the `change_page_attr_set_clr()` kernel function which will explicitly call `vm_unmap_aliases()` leading to a purge. This is extremely useful for improving reliability by starting from a known clean state.

Reclaiming the UAF

The first step in the exploitation is to gain control of the UAF. The goal is to trigger it, provoke a large number of purges so that the affected kernel stack is actually freed and finally allocate a v4l2 buffer that overlaps the UAF address. By memory mapping (via `mmap`) this buffer, we can get full control over the UAF area. First, we begin by allocating deviceless memory in the NVIDIA driver until there is no unpurged area left and the pools are empty. Then, we can use the forking primitive to fill all the holes in the `vmalloc` area. This will ensure a clean state where future allocations will be made one right after the other even if they are of different sizes. When forking, we will make most of the processes terminate immediately. However, some of them will be kept alive at regular intervals, to create gaps that are smaller than the v4l2 buffers we will allocate later. This way, even after the unpurged stacks are freed (red allocations in the next figure), any v4l2 buffer allocated will end up in the clean area, while smaller allocations on the system that could disrupt the exploitation will end up in these holes. We will refer to the kept alive stacks as guards.



Once we reach the clean state, we do the final setup by:

Forking and keeping alive a "beacon" process (used later)

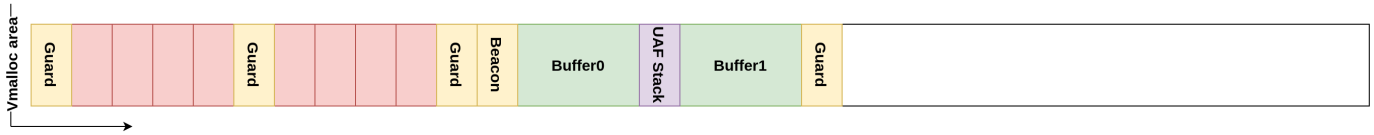
Allocating and freeing a medium-sized v4l2 buffer

Forking a new process and triggering bug #1 with it

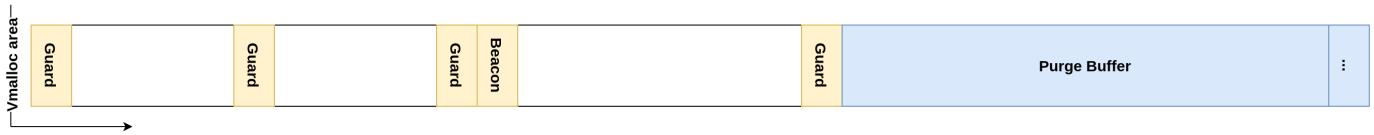
Allocating and freeing a medium-sized v4l2 buffer again

Allocating and keeping alive a final guard process

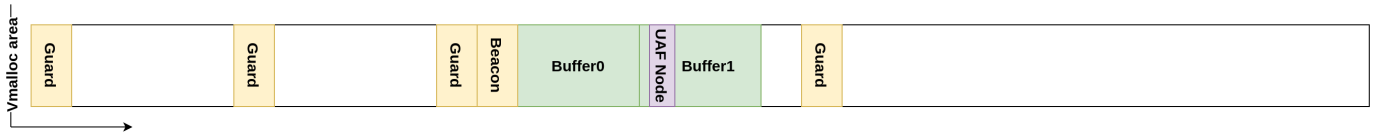
These steps are very time sensitive as any other allocation on the system may get in between, most probably leading to a failure of the exploitation.



After that, it's possible to monitor the oops happening by waiting for the triggering process to get killed. Once it happened, the driver will be in a reduced state. Indeed, the kernel thread that hit the bug was killed while holding locks, so, most new calls to the drivers will just hang indefinitely. This means we can't use the side effect purge method and instead have to use large v4l2 buffers. These large allocations will not interfere with the area of the UAF as they will be allocated further away because of the guard stacks.



Once we allocated and freed enough of these large allocations so that the pools are empty, we can just allocate a set of two medium-sized v4l2 buffers. These buffers will be backed by only one `vmalloc` allocation and so they will be one after the other. If everything went right, they should end up being allocated just after the beacon process because of guards. The second buffer will contain the UAF. The reason we used two buffers is because Buffer0 will be used later in the exploitation for data storage.



The tree data structure

The UAF we now control somewhere in Buffer1 is the node of a [binary Red/Black tree]. It serves as the underlying data storage for a map container, the global `threadStateDatabase.dbRoot`. This map is used to store structures of type `THREAD_STATE_NODE` in the time frame between `threadStateInit()` and `threadStateFree()`. The implementation is intrusive so every `THREAD_STATE_NODE` structure contains a `struct MapNode` defined as follows:

```
// src/nvidia/inc/libraries/containers/map.h

struct MapNode {
    NvU64       key;
    MapNode    *pParent;
    MapNode    *pLeft;
    MapNode    *pRight;
    NvBool      bIsRed;
};
```

This data structure will be our primary focus. The `THREAD_STATE_NODE` structure also contains interesting fields such as function pointers. However, the `threadStateInit()` and `threadStateFree()` functions only perform operations on the structure found in their own stack, so that it' not possible to trick them into calling these function pointers on a node coming from the tree.

Revealing kernel memory addresses

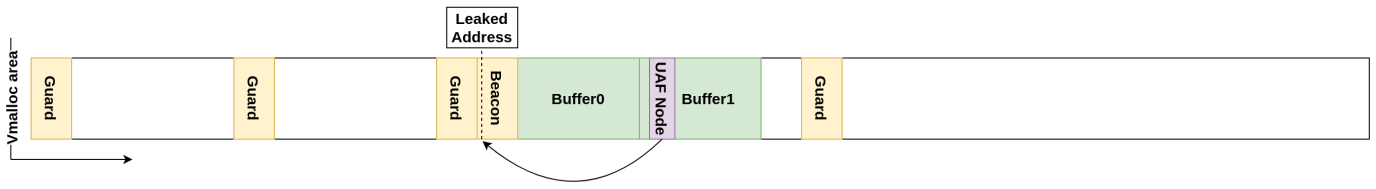
Even if the driver is in a reduced state, one operation still working is opening a GPU device (e.g. `/dev/nvidia0`). Fortunately, this triggers a call to `rm_is_device_sequestered()` which uses the `threadStateInit()` and `threadStateFree()` combo. This means a new node will be inserted and removed from the tree each time we open the device file. As the nodes have a very short life span, we can expect the UAF node to be the only one in the tree. As such, the UAF node will be the root and we can expand the tree by creating our own node linked to it. Before doing that, we need to solve two problems:

Where is the UAF node located in Buffer1 to be able to modify it

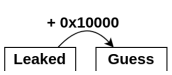
What is the address of Buffer0 so we can create our own nodes inside it and link them together

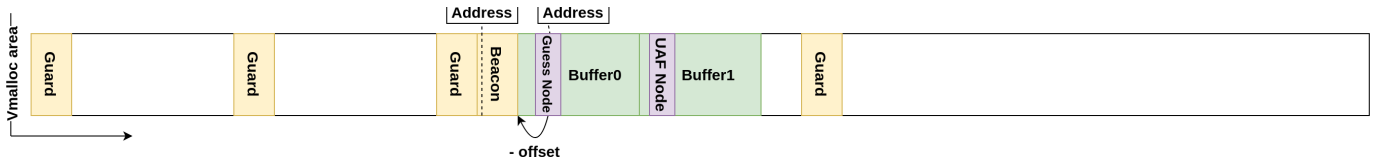
Because of `random_kstack_offset`, we can't predict the offset of the UAF node in the stack and so its offset in Buffer1. Fortunately, a zeroed out `struct MapNode` is a valid node (a black node with no children). Therefore, if the whole Buffer1 is zeroed out, insertions in the tree can happen without any issue. Because the key will also be `0`, new nodes will be inserted as the right child of the UAF node. So, when calling `open` on the GPU device, `node.pRight` will very briefly be filled with a pointer to a child. To find the offset of the node in Buffer1, a possibility is to call `open` repeatedly from another process and scan Buffer1 until we find a non-zero value.

Furthermore, because `node.pRight` will point to the `struct MapNode` stored in the stack of the process calling `open`, it's effectively leaking an address inside its kernel stack. We set up a beacon process for this reason, ensuring its stack is positioned just before Buffer0.



Once the beacon stack address is leaked, we can guess an address that should be part of Buffer0. If we set `node.pRight` of the UAF node to this guessed address, new nodes will be inserted as the right child of the guessed node. By calling `open` repeatedly again and scanning Buffer0 for a nonzero value, we can find the offset of the guessed node. By subtracting the found offset to the guess address we ultimately find the exact kernel address of Buffer0.





The guess address technique may seem superfluous, but it's essential as we cannot ascertain the exact beacon stack base address from the leak. This ambiguity is due to the `random_kstack_offset` feature and the possibility that a kernel stack allocation can begin at any page boundary.

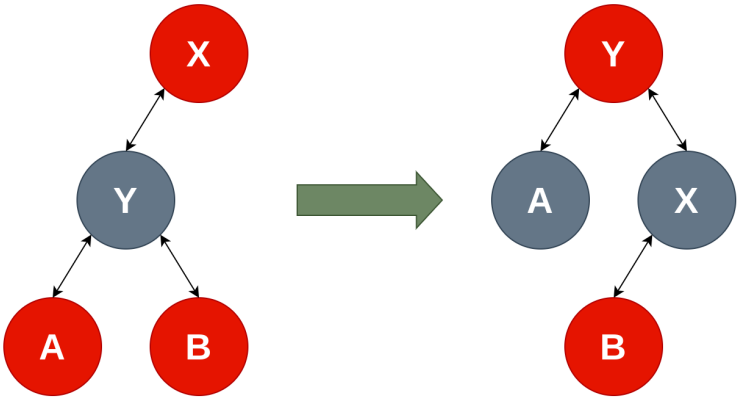
A first write primitive

Now that we have everything needed to create arbitrary trees, we need to find arrangements that could lead to interesting primitives during either insertion or deletion of a node. These operations always comprise the actual addition or removal of the node in the tree followed by a fixup phase (`_mapInsertFixup()` or `_mapDeleteFixup()`). These fixup functions will usually recolor and perform rotations in the tree. They are interesting as they loop up through it allowing us to have at least a bit of control on the execution. The goal is then to trick them into reading or writing at an arbitrary address. To do so we can use part of the rotation code:

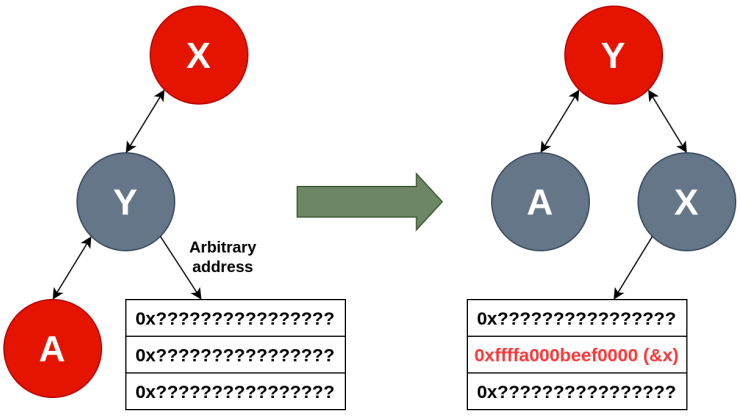
```
static void _mapRotateRight
(
    MapNode **pPRoot,
    MapNode *x
)
{
    // rotate node x to right
    MapNode *y = x->pLeft;
    // establish x->pLeft link
    x->pLeft = y->pRight;

    if (y->pRight)
        y->pRight->pParent = x; // <= Here is the only use of y->pRight

    // establish y->pParent link
    y->pParent = x->pParent;

    if (x->pParent)
    {
        if (x == x->pParent->pRight)
            x->pParent->pRight = y;
        else
```

```
            x->pParent->pLeft = y;
    }

    else
        (*pPRoot) = y;

    // link x and y
    y->pRight = x;
    x->pParent = y;
}
```

There is a mirror version of this code (`_mapRotateLeft`) that could also be used, but we will focus on the right one. When executed this function will set `pParent` in the node pointed to by `y->pRight` if it's not null without ever using it again. Visually the rotation looks like this:



If we set `y->pRight` to an arbitrary address, we can obtain a constrained arbitrary write primitive because a pointer to `x` will be written to `y->pRight + offsetof(MapNode, pParent)`.



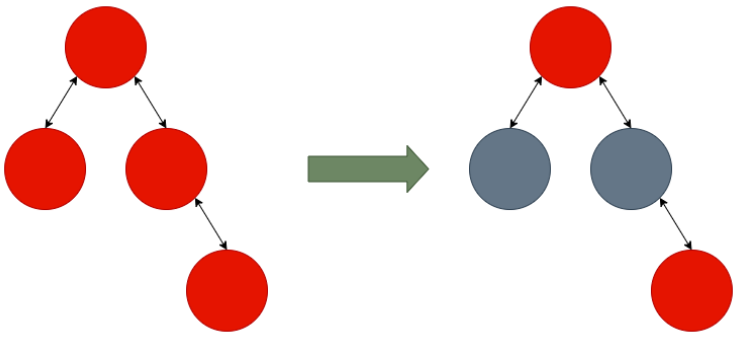
Assuming `x` is one of our nodes in Buffero, we can consider that we are writing a pointer to a controlled address. The right rotation can be attained from `_mapInsertFixup()` without the value of `y->pRight` being used by building the right tree structure. There might be better primitives available directly from the tree but this one have the advantage of being straightforward and reliable.

Selecting a target

Next step is to find what exactly to overwrite. Without relying on other bugs, we are only aware of a few addresses allocated by `vmalloc`. One solution would be to shape the `vmalloc` area so that an interesting allocation is found close to our beacon and buffers in order to guess its address. That should be doable, but after searching for a bit, I didn't find any interesting structure. As a matter of fact, `vmalloc` is not used that much in the kernel and mostly for big buffers because of its page granularity. Also, there are in fact multiple separated `vmalloc` areas, limiting the possibilities.

Instead, targeting kernel stacks seemed easier as we already know we can leak their addresses. We used this capability before to guess the address of Buffer0. However, we can also leak the address of other interesting values in the stack during the execution of `open` (the syscall that triggers the insertion in the tree). Indeed, offsets in the stack should be constant for a given kernel and driver binaries, we can just calculate beforehand the distance between the node and a specific value we want to target in the stack. The use of `kstack_random_offset` changes nothing, as the offset is added before the syscall is executed.

However, in order to use this method combined with the write primitive, the target address needs to be computed in the very small time frame between the insertion of the node and the rotation of the tree that will trigger the write. This is due to the address changing every syscall because of `kstack_random_offset`. By default, there is not enough time for the userland process to modify the mapped memory in time. However, we can artificially increase the time taken by the tree iteration before the rotation is executed. The `_mapInsertFixup()` function has a recolor-only path which will perform the following:

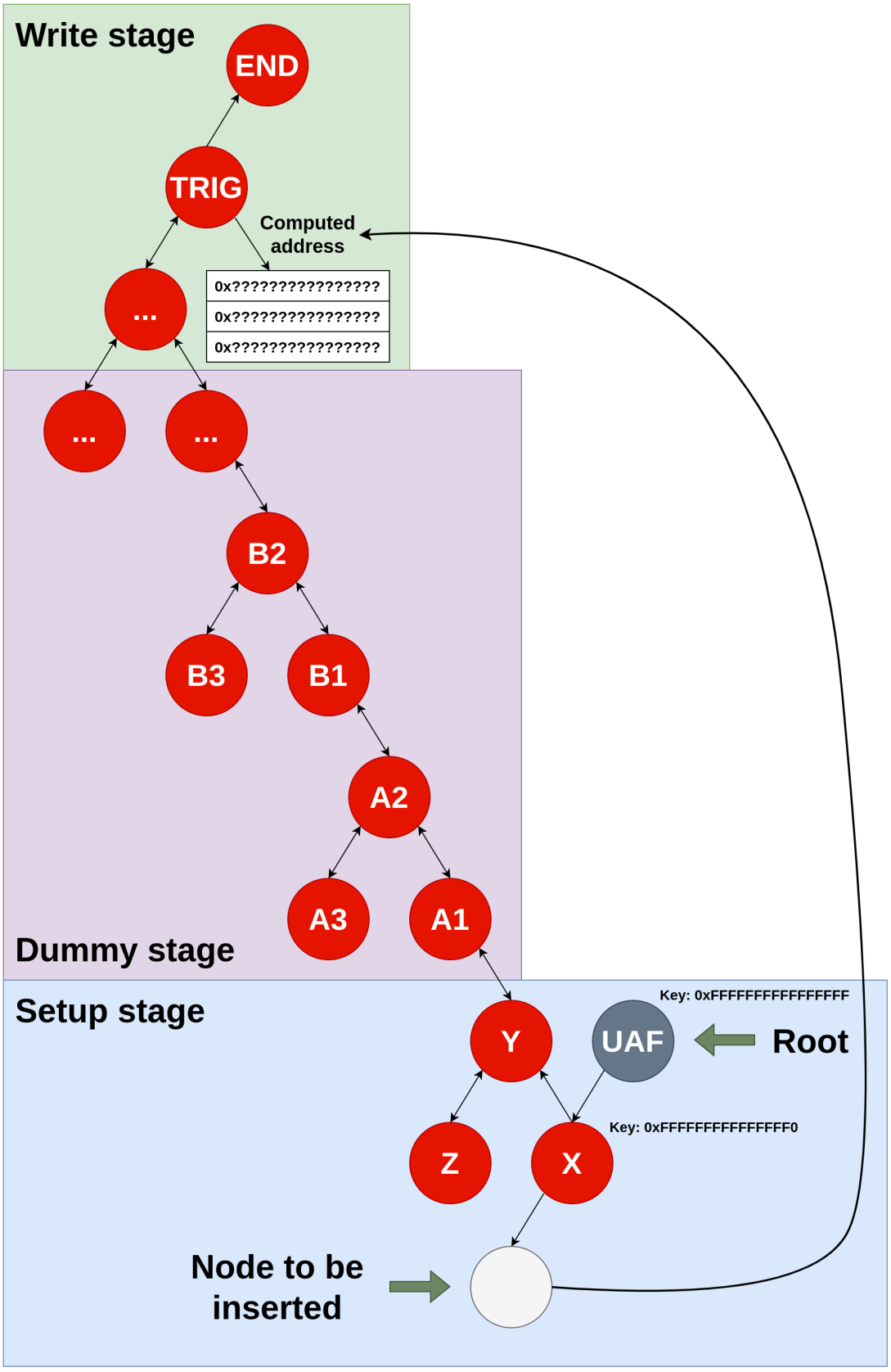


For our purposes, recoloring has no side effects and can be used to waste time, by building a tree using the pattern found in the previous figure. We can then build a three-staged tree:

Setup: Welcomes the new node insertion and make the iteration jump into an alternate part of the tree (i.e. that is not under the root) using a flawed `pParent` pointer

Dummy: Combination of an arbitrary number of recolor patterns used to waste time (256 patterns were used for the proof of concept)

Write: Perform a write using a rotation, the address will be computed and filled in dynamically by userland



The node is made to be inserted as a left child using very large keys to facilitate the jump into the dummy phase. This tree allows to reliably write a pointer to controlled data over any chosen value in the kernel thread stack during the handling of the `open` syscall. The written data will effectively be a pointer to the node labeled `END`. After the rotation, we are free to write any data at this address.

Escalating with stack corruption

Now, we just need to find a good candidate pointer to overwrite. A very interesting one is the `file` pointer in `path_openat()`:

```
// fs/namei.c

static struct file *path_openat(struct nameidata *nd,
            const struct open_flags *op, unsigned flags)
{
    struct file *file;
    int error;

    file = alloc_empty_file(op->open_flag, current_cred()); // struct
file allocation
    if (IS_ERR(file))
        return file;

    if (unlikely(file->f_flags & __O_TMPFILE)) {
        error = do_tmpfile(nd, flags, op, file);
    } else if (unlikely(file->f_flags & O_PATH)) {
        error = do_o_path(nd, flags, file);
    } else {
        const char *s = path_init(nd, flags);
        while (!(error = link_path_walk(s, nd)) &&
               (s = open_last_lookups(nd, file, op)) != NULL)
            ;
        if (!error)
            error = do_open(nd, file, op); // function that will lead to
the write
        terminate_walk(nd);
    }
    if (likely(!error)) {
        if (likely(file->f_mode & FMODE_OPENED))
            return file;
        WARN_ON(1);
        error = -EINVAL;
    }
    fput_close(file);
    if (error == -EOPENSTALE) {
```

```
        if (flags & LOOKUP_RCU)
            error = -ECHILD;
        else
            error = -ESTALE;
    }
    return ERR_PTR(error);
}
```

When looking at the compiled binary for the target version, we can see that the `file` pointer is stored in `r12`. The `do_open()` function spills `r12` on the stack and at the same time will lead to the call that triggers our write. Meaning that we can ultimately overwrite the `file` pointer to make it point into our memory mapped Buffer0 by precomputing the offset between `struct MapNode` and the spilled `r12` register in the stack. This modified file pointer will be returned by `path_openat()` and associated with a file descriptor in the calling process by `fd_install()` in `do_sys_openat2()`. There are a few checks and dereferences that may cause issues, but by creating a fake `struct file` with somewhat sensible values it's possible to overcome them easily.

It's to be noted that the `file` structure is defined with the `__randomize_layout` macro. This will lead to the fields being out of order and that we have to find the offsets for the specific target kernel. Fortunately, in our case, these can be easily extracted from the Ubuntu debug packages.

Leaking KASLR

The control over a `struct file` is extremely powerful. This structure notably contains several function pointers due to the Virtual File System layer. However, our last barrier to a full exploitation is KASLR (Kernel Address Space Layout Randomization). To break it, we can leverage some syscalls that check the type of a file by comparing the `f_op` pointer to the expected `struct file_operations`. For example, `recvfrom` uses `sock_from_file()` to get access to private data specific to sockets and checks the file type using the `f_op` pointer:

```
// linux/net/socket.c

struct socket *sock_from_file(struct file *file)
{
    if (likely(file->f_op == &socket_file_ops))
        return file->private_data;  /* set in sock_alloc_file */

    return NULL;
}
```

If the pointers don't match and `sock_from_file()` returns null, `recvfrom` will simply return `-ENOTSOCK`. So, we can call this syscall repeatedly on the file descriptor linked with our controlled

`struct file`, starting with `f_op` set to the static address of `socket_file_ops` and then incrementing it to test all the possible slided values. KASLR is leaked when the syscall returns something other than `-ENOTSOCK`. This is a somewhat fast process due to KASLR entropy only being 9 bits.

Wrapping up

After that, we can just create our own file operations table. I decided to use the `llseek` handler to perform arbitrary functions calls in the kernel. It's defined as follows:

```
loff_t (*llseek) (struct file * file, loff_t offset, int whence);
```

It's interesting because the syscall handler does not perform any check on the file before calling the handler. Also, we have control and access to all the parameters and the return value directly from userland. The limitations are as follows:

The `whence` parameter should be less than five

The first parameter is a pointer to our controlled `struct file` meaning we must input or output arbitrary data from the start of the structure. That's not a problem on the target version because all the fields in the start are unused, but it could be if we are very unlucky with the randomized order of the fields.

By setting the handler to point to selected kernel functions and then calling the `llseek` syscall, we can build a basic set of primitives:

Kernel symbolication with `unsigned long kallsyms_lookup_name(const char *name)`

Kernel arbitrary read with `void *memcpy(void *dest, const void *src, size_t count)`

Kernel arbitrary write with `int debugfs_u64_get(void *data, u64 *val)`

For testing them, we can escalate the privileges of our userland process. We just need to symbolicate `init_task` and iterate the tasks until we find the one corresponding to our process. Then, we can overwrite the creds to become root and open a shell. Below is the full proof of concept running in real time:

To conclude, a couple of key points to consider. First, the exploit is sensitive to system activity, particularly forking and calls to the NVIDIA driver during specific time frames. This poses a challenge on systems under constant heavy load where the exploitation will most likely fail.

Second, as previously mentioned the kernel oops triggered by bug #1 causes multiple locks to be held, rendering most of the NVIDIA driver unusable. It should be possible to manually unlock the driver using the kernel read and write primitives, but this has not been tested.

The complete proof-of-concept exploit described in this blog post is available here

Disclosure timeline

Below we include a timeline of all the relevant events during the coordinated vulnerability disclosure process with the intent of providing transparency to the whole process and our actions.

2025-06-18 Quarkslab reported the vulnerabilities to NVIDIA PSIRT.

2025-06-18 NVIDIA acknowledged the report and asked if we planned to disclose the bugs.

2025-06-25 Quarkslab replied that we planned to publish a blog post or conference talk but there was no specific plan and it would be determined along the coordination process.

2025-06-26 NVIDIA acknowledged last email and promised to keep us updated as the process evolves.

2025-07-14 NVIDIA indicated it couldnt reproduce the bugs.

2025-07-21 Quarkslab sent a reply to NVIDIA noting that the report had specific comments about triggering the bugs and exploitability.

2025-07-22 NVIDIA acknowledged the last communication and said it was passed to the dev

team.

2025-07-24 Quarkslab sent further details about how to reproduce the bugs and asked what runtime environment was NVIDIA using to try to repro them.

2025-07-28 Quarkslab re-sent the prior email with a minimized PoC.

2025-08-08 NVIDIA provided information about their runtime environment, the internal case numbers, and said they will implement the fixes by mid-january 2026, and asked if Quarkslab could delay disclosure until then.

2025-08-11 NVIDIA reiterated the request to postpone disclosure until mid-January 2026.

2025-08-12 Quarkslab replied that the bugs were first reported in June 18th and mid-January was well past the standard 90 day normally agreed for coordinated disclosure and that we did not see a rationale for postponing publication by, at a minimum, 3 months. Therefore Quarkslab continued with the publication deadline set to September 23rd 2025 and offered to extend the deadline an additional 30 days provided NVIDIA gave us some insights about the full scope of affected products and if the fixes are to be released as a stand alone security fix, as opposed to rolled into a version bump that includes other code changes.

2025-08-12 NVIDIA acknowledged our email and said it will communicate the deadline to the product team.

2025-08-14 NVIDIA provided an update and requested the 30-day extension offered. Indicated the fix for the null pointer dereferrence bug, which would make the UAF not reachable, was under review. The team was determining whether the fix would be a standalone update or included in a regular version update release. NVIDIA said it would be happy to share the final disclosure security bulletin language before releasing it to partners and the public.

2025-08-18 NVIDIA requested confirmation of the 30 day extension to the disclosure deadline.

2025-08-18 Quarkslab agreed to extend the disclosure deadline to October 23rd 2025.

2025-10-09 NVIDIA published Security Bulletin: NVIDIA GPU Display Drivers - October 2025 crediting CVE-2025-2330 to Quarkslab.

2025-10-09 Quarkslab asked NVIDIA when they planned to fix the UAF bug or if it was the fix for CVE-2025-23280 in the October update, which was not credited to anyone.

2025-10-09 NVIDIA apologized for not having notified Quarkslab of the security bulletin release and said it would correct the attribution of CVE-2025-23280, which was indeed the Kernel UAF bug.

2025-10-14 This blog post is published.

If you would like to learn more about our security audits and explore how we can help you, get in touch with us!