PTM - Page Table Manipulation From Usermode

Author: _xeroxz — Email: _xeroxz@back.engineer — Website: back.engineer

December 1, 2020

1 Abstract

PTM is a Windows 10 C++ library that allows a programmer to manipulate all memory, physical, and virtual from user-mode. The project inherits an interface from VDM allowing the use of a physical memory read-write primitive to fuel this project. VDM is used solely to configure the page tables in such a way that PTM can manage them from user-mode. Once the page tables are configured for PTM VDM is no longer required. However, VDM can inherit an instance of PTM as a means to read and write physical memory. Both VDM and PTM work extremely well together and independently from each other.

2 Introduction

Page table manipulation is an extremely powerful primitive. One that allows groundbreaking projects to be created such as patching the kernel only in specific process-contexts, or mapping of a source process address space into a target process address space. PTM is a user-mode library that allows a programmer to manipulate page tables from user-mode on 64-bit Windows 10 systems. PTM does this by using VDM; a project designed to abuse vulnerable drivers exposing a physical memory read-write primitive (RWP) to elevate to arbitrary kernel execution. VDM is used to configure the page tables in such a way that they can be managed from user-mode without the need for VDM's vulnerable driver being loaded into the kernel after initialization. PTM can then be used to get and set all levels of page table entries, translation linear virtual addresses

from user-mode, map physical memory into virtual memory, and even create new page tables. PTM can also be used as a means to directly read and write physical memory, thus it can be used with VDM to leverage arbitrary kernel execution without the need of VDM's vulnerable driver being loaded into the kernel.

3 Basics of Virtual Memory and Paging

Paging is the concept of breaking memory into fixed-sized chunks called pages. Pages can be moved in and out of physical memory allowing for memory that is not accessed frequently to be moved to disk. In order for this to work, the CPU cannot directly interface with physical memory instead the CPU interfaces with virtual memory. Virtual addresses are translated to physical addresses using a set of tables called page tables. On a 64-bit system with the CPU in long mode, there are four layers of page tables: PML4(s), PDPT(s), PD(s), and lastly PT(s). All page tables are the same size (1000h bytes) unless configured otherwise. Each page table entry is eight bytes in size. This means that each table contains 512 entries (8 * 512 = 1000h). The last twelve bits of every virtual address is called the page offset and is an offset into a physical page. The page offset of a virtual address can be bigger than 12 bits depending on the paging structure configuration for a given virtual address. The length of the page offset field can be either 12 bits (physical page is 4kB), 21 bits (2MB physical page), or 30 bits (1GB page).

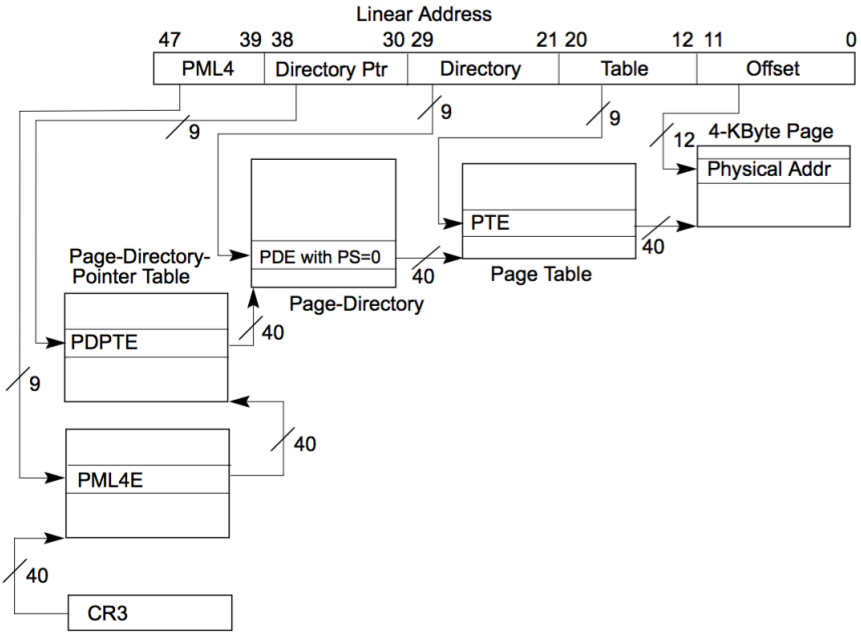

Figure 1: Virtual Address Translation

In order to translate linear virtual addresses to linear physical addresses, the page tables must be traversed. As depicted in figure one, each virtual address space has its own PML4, the physical address of this table is stored in CR3.

4 Interfacing With Page Tables on Windows

On Windows, the thread scheduler utilizes KPROCESS.DirectoryTableBase when scheduling threads. The KPROCESS structure is a substructure of the EPROCESS structure and contains DirectoryTableBase at offset 28h. A programmer using VDM can obtain the linear physical address of the PML4 of a process easily by DKOM'ing a desired process KPROCESS structure.

```
kd> dt !_KPROCESS ffffc38759d9e080
nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 ProfileListHead  : _LIST_ENTRY
   +0x028 DirectoryTableBase : 0x00000001`15684000
   +0x030 ThreadListHead   : _LIST_ENTRY
   +0x040 ProcessLock      : 0
   +0x044 ProcessTimerDelay : 0
   +0x048 DeepFreezeStartTime : 0
```

Once the physical address of the desired processes PML4 has been obtained the trick is interfacing with the paging structures. Although VDM allows reading and writing of physical memory, be aware that MmMapIoSpace cannot be used to map the paging structures into virtual memory. Drivers that use MmCopyMemory and ZwMapViewOfSection to interface with physical memory can however be used to directly manipulate the page tables. To properly support VDM which PTM inherits as a codebase, the project does not rely on the physical read and write primitive exposed from the driver. Instead PTM allocates its own set of page tables and inserts a PML4E into the current processes PML4 pointing at such tables. This allows a programmer to map physical memory at will into the current virtual memory address space, all from user-mode. In other words, once the tables are allocated and configured, there is no need for VDM anymore since the paging tables can be controlled entirely from user-mode.

5 Translation Look-aside Buffer

The translation look-aside buffer is a hardware-based cache that assists in translating linear virtual addresses to linear physical addresses. The TLB caches virtual to physical address translations, as well as other information like page access rights and cache type information. Although extremely important for efficiency, the TLB has made PTM an interesting challenge. For example, when physical memory is mapped into a virtual address space, page table entries will be inserted, or changed. This insertion or alteration of an existing page table

entry may be of a cached entry in the TLB. This means that the effects applied to the page table entry will not be seen until the TLB entry for the given virtual page has been invalidated, along with the changes written to main memory. To counteract this, the CPU has an instruction that allows a programmer to invalidate a page table entry in the TLB's cache. This instruction is called INVLPG and is a privileged instruction. It's not something PTM can use since the library is designed to operate entirely from user-mode. Directly invalidating TLB is not the only way to invalidate entries. If a page fault occurs, the TLB invalidates entries for the given address that caused the fault (the address in CR2). This is an effective method for invalidating desired virtual addresses from user-mode but is extremely slow. Context switches do not inherently cause the TLB to flush, rather the PCID is changed to another PCID. This allows the TLB to retain entries from multiple address spaces and improve performance. However, yielding execution can invalidate TLB entries because the scheduler will reschedule the logical processor to execute somewhere else for some time, possibly filling the TLB with other entries and removing the ones that were previously cached.

5.1 TLB - Outrun

Although the TLB is an effective hardware-based cache, it cannot cache linear virtual addresses that have not been accessed before, this simple fact means a programmer can create a new linear virtual address every single time they would want to map a new physical page into virtual memory. This, however, is not a solid solution that works soundly on all modern CPUs. With the industry pushing forward with virtualization technology, the expansion of the TLB continues. Thus solely generating a new linear virtual address every time you would want to interface with a physical page is not a sound solution and is already unstable on most modern AMD chips. Instead combining this technique with other techniques is ideal.

```
auto ptm_ctx::map_page(void* addr) -> void*
{
    ++pte_index;
    if (pte_index > 511)
    {
        ++pde_index;
        pte_index = 0;
    }

    if (pde_index > 511)
    {
        ++pdpte_index;
        pde_index = 0;
    }

    if (pdpte_index > 511)
        pdpte_index = 0;

    // insert paging table entries down here
```

```
    //... (refer to PTM repo to see that code)...
    // returns the newly generated virtual address...
    return get_virtual_address();
}
```

The code above generates a new linear virtual address that has not been accessed before. This linear virtual address points to the requests physical page in memory. This allows the programmer to circumvent the TLB by accessing new linear virtual addresses instead of trying to invalidate TLB entry of an existed and already cached page. This however has limitations since the code only provides 512^3 different possible virtual pages.

5.2 TLB - Benefit of The Doubt

Although outrunning the TLB is the fastest solution for mapping physical memory into virtual memory without needing to invalidate any TLB entries, it is not the most stable on modern hardware. Instead, a mixture of generating a new virtual address and an SEH try/except loop is preferred. By giving the new virtual address the benefit of the doubt that it has not been cached yet, an attempt to access the newly created page is performed. If the access is successful, the new linear virtual address is returned to the caller of ptm::ptm_ctx::map_page. However, if the access causes a page fault, the TLB invalidates the entries associated with this newly created linear virtual address. The except block then attempts to access the new page in a loop whilst yielding execution at each failure to access the new virtual address. This technique provides the most performant solution to dealing with the TLB from user-mode. This method guarantees that the linear virtual address generated is accessible before returning it to the caller.

```
auto ptm_ctx::get_virtual_address() const -> void*
{
    //...

    // start off by making sure that
    // the address is accessible...
    __try
    {
        *(std::uint8_t*)new_addr.value = *(std::uint8_t*)
            new_addr.value;
        return new_addr.value;
    }

    // if its not accessible then the
    // TLB just invalidated its entry...
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        // loop until this new address is accessible
        // do not return until this new virtual
        // address is accessible....
```

```
        while (true)
        {
            // try again to access the page again
            // and it should return...
            __try
            {
                *(std::uint8_t*)new_addr.value =
                    *(std::uint8_t*)new_addr.value;
                return new_addr.value;
            }

            // if its still not accessible, then we are
            // going to let the core get
            // rescheduled for a little...
            __except (EXCEPTION_EXECUTE_HANDLER)
            {
                while (!SwitchToThread())
                    continue;
            }
        }
    }
    return new_addr.value;
}
```

This code above is executed every single time a new page is mapped into virtual memory via PTM. It ensures that the new virtual address created is accessible before returning the new virtual address.

6 Complications With Paging

Although PTM manages its own set of paging tables from user-mode the physical address of these VirtualAlloc'ed page tables can change. The page tables allocated by VirtualAlloc can themselves be paged to disk if they are not accessed frequently. When they are paged back in when accessed again it is highly likely that they will be allocated in a new physical page. The working set manager plays a large part in paging memory to disk. This thread loops indefinitely and is created when the operating system starts. This working set manager thread has proven to be quite annoying and thus it is suspended by PTM.

```
auto get_setmgr_pethread(vdm::vdm_ctx& v_ctx) -> PETHREAD
{
    ULONG return_len = 0u;
    std::size_t alloc_size = 0x1000u;
    auto process_info =
        reinterpret_cast<SYSTEM_PROCESS_INFORMATION*>(
            malloc(alloc_size));

    while (NtQuerySystemInformation
```

```
    (
        SystemProcessInformation,
        process_info,
        alloc_size,
        &return_len
    ) == STATUS_INFO_LENGTH_MISMATCH)
        process_info =
        reinterpret_cast<SYSTEM_PROCESS_INFORMATION*>(
            realloc(process_info, alloc_size += 0x1000));

    const auto og_ptr = process_info;
    while (process_info &&
        process_info->UniqueProcessId != (HANDLE)4)
        process_info =
            reinterpret_cast<SYSTEM_PROCESS_INFORMATION*>(
                reinterpret_cast<std::uintptr_t>(
                    process_info)
                    + process_info->NextEntryOffset);

    auto thread_info =
        reinterpret_cast<SYSTEM_THREAD_INFORMATION*>(
            reinterpret_cast<std::uintptr_t>(process_info)
                + sizeof SYSTEM_PROCESS_INFORMATION);

    static const auto ntoskrnl_base =
        util::get_kmodule_base("ntoskrnl.exe");

    const auto [ke_balance_um, ke_balance_rva] =
        util::memory::sig_scan(
            KE_BALANCE_SIG, KE_BALANCE_MASK);

    auto rip_rva =
        *reinterpret_cast<std::uint32_t*>(ke_balance_um +
            19);

    const auto ke_balance_set =
        ntoskrnl_base + ke_balance_rva + 23 + rip_rva;

    const auto [suspend_in_um, suspend_rva] =
        util::memory::sig_scan(
            SUSPEND_THREAD_SIG, SUSPEND_THREAD_MASK);

    rip_rva =
        *reinterpret_cast<std::uint32_t*>(
            suspend_in_um + 1);

    const auto ps_suspend_thread =
        reinterpret_cast<void*>(
            ntoskrnl_base + rip_rva + 5 + suspend_rva);

    static const auto lookup_pethread =
        util::get_kmodule_export(
            "ntoskrnl.exe", "PsLookupThreadByThreadId");
```

```
    for (auto idx = 0u; idx < process_info->NumberOfThreads;
         ++idx)
    {




        if (thread_info[idx].StartAddress ==
            reinterpret_cast<void*>(ke_balance_set))
        {
            PETHREAD pethread;
            auto result = v_ctx.syscall<
                PsLookupThreadByThreadId>(
                 lookup_pethread, thread_info[idx]
                     .ClientId.UniqueThread, &pethread);

            free(og_ptr);
            return pethread;
        }
    }

    free(og_ptr);
    return {};
}
```

The code defined above will obtain the PETHREAD of the working set manager using VDM and return it to the caller. The caller can then use VDM to syscall into PsSuspendThread to suspend the working set manager thread.

```
auto stop_setmgr(vdm::vdm_ctx& v_ctx, PETHREAD pethread) ->
    NTSTATUS
{
    static const auto ntoskrnl_base =
        util::get_kmodule_base("ntoskrnl.exe");

    const auto [suspend_in_um, suspend_rva] =
        util::memory::sig_scan(
            SUSPEND_THREAD_SIG, SUSPEND_THREAD_MASK);

    const auto rip_rva =
        *reinterpret_cast<std::uint32_t*>(
            suspend_in_um + 1);

    const auto ps_suspend_thread =
        reinterpret_cast<void*>(
            ntoskrnl_base + rip_rva + 5 + suspend_rva);

    return v_ctx.syscall<PsSuspendThread>(
        ps_suspend_thread, pethread, nullptr);
```

```
}
```

7 Conclusion

To conclude PTM is yet another highly modular library that allows a programmer to manipulate all memory, virtual and physical from user-mode. PTM is used in all of my other recent projects such as reverse-injector, PSKP, PSKDM, and kmem. Although this project is ideal for page table manipulation, it does have its drawbacks. As described in section six, the VirtualAlloc'ed page tables can be paged to disk themselves and thus when paged back into physical memory they will most likely be allocated in a new physical page. However, this has proven to be an insignificant issue after suspending the working set manager thread which is responsible for paging virtual memory to disk.

7.1 Related Work

Reverse-Injector

PSKDM (Process-Context Specific Kernel Driver Mapper)

PSKP (Process-Context Specific Kernel Patches)

pclone (Process Cloning)

/proc/kmem (Reimplementation of it for Windows 10)

Hyperspace (Creating an address space not associated with a Windows process)