Exploit Two Xen Hypervisor Vulnerabilities

Shangcong Luan

Cloud Platform Security Team of Alibaba Cloud

shangcong.lsc@alibaba-inc.com

Abstract

The Xen Project is a widely used virtualization platform powering some of the largest clouds in production today. In the process of virtualization security research on it, our team has discovered two critical vulnerabilities in the PV mode Memory Management of Xen Hypervisor. This paper aims to present a comprehensive study of Xen Hypervisor PV Guest Memory Management and detail our two critical vulnerabilities. Furthermore, full exploitation technologies will be discussed.

Keywords: Xen Security, XSA-148, Dome Breaking, XSA-182, Ouroboros, hypervisor exploitation, VM Escape

1. Introduction

Xen is an open source project providing virtualization services that allow multiple computer operating systems to execute on the same computer hardware concurrently. It originated as a research project at the University of Cambridge and the first public release was made in 2003. Since then the project has attracted extensive attention from virtualization and security researchers. In the past decades, virtual machine escape was considered as an unreal story because of the complex and effective isolation supported by virtualization technologies although some security vulnerabilities had been found. But unfortunately, a SVGA emulation bug was reported on VMware in 2008 and at the next year's Blackhat conference, researchers from Immunity Team disclosured a fully working exploit which proved virtual machine escape isnt a joke. In 2012, the unbelievable SYSRET vulnerability was disclosured and Xen was affected at this time. In 2015, the infamous VENOM vulnerability in QEMU evoked worldwide repercussions although no one could exploit it in the real scene.

There is no doubt that virtual machine escape is a real threat against virtualization security. Thus our team started the research last year and the Xen project was our main target. After some months, we found a series of risks and vulnerabilities which contained two critical logic issue: XSA-148 and XSA-182. The XSA-148 vulnerability was reported in October 2015. The XSA-182 vulnerability was intended to be the theme of our presentation at Blackhat, but it was reported in mid July by Quarkslab Team while we were writing this paper. The two vulnerabilities in XSA-148 and XSA-182 are both exploitable in reality and will be discussed in this paper.

The important content of this paper is arranged as follows:

- Section 2 will give an overview of Xen architecture.
- Section 3 will analyze details of the two vulnerabilities.
- Section 4 will discuss all kinds of exploitation technologies.

2. Xen Basis

This section gives a high level architectural overview of Xen and contains only basic infomation about the project. For a more complete description of its architecture please reference offical documents.

2.1. Xen Hypervisor

Xen Hypervisor is the basic abstraction layer of software that sits directly on the hardware below any operating systems. It is responsible for CPU scheduling and memory partitioning of the various virtual machines running on the hardware device. It is a type-1, native or bare-metal hypervisor and has no knowledge of networking, external storage devices, video, or any other common I/O functions founded on a computing system.

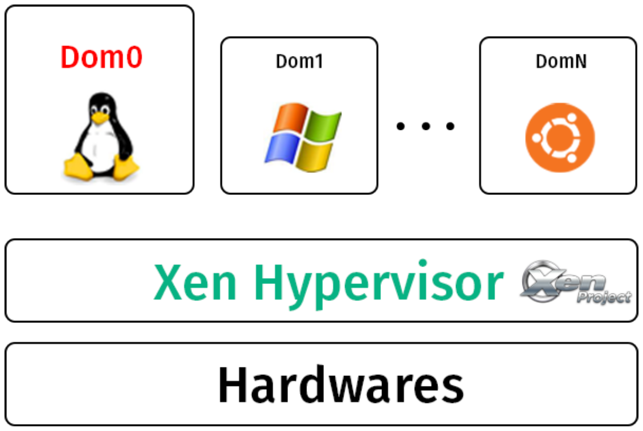


FIGURE 1. Xen Architecture

2.2. Guest Domain

The virtual machine running on the Xen Hypervisor is called guest domain. From the perspective of virtualization approach, guest domains could be simply grouped into two main types: PV mode guest domains and HVM mode guest domains. Compared to PV mode guest domains, HVM mode guest domains are virtualized with Hardware-assisted virtualization technologies supported by modern hardwares.

2.3. Hypercall

A hypercall is a software trap from a domain to the hypervisor, just as a syscall is a software trap from an application to the kernel. Domains will use hypercalls to request privileged operations like updating pagetables.

2.4. Memory Management

For PV mode guest domains, Xen Hypervisor manage their memory using Direct Paging mechanism which allows MMU access guest page tables directly.

In this case, guest page tables must be validated before they are visiable to MMU. Xen Hypervisor is responsible for registering guest page tables directly with the MMU, and restrict guest domains to read-only access. Page table updates are passed to Xen via hypercalls to ensure safety and requests are validated before being applied.

To aid validation, the hypervisor associate a type and reference count with each machine page frame. A frame may have any one of the following mutually-exclusive types at any point in time: page directory (PD), page table (PT), local descriptor table (LDT), global descriptor table (GDT), or writable (RW). Note that a guest OS may always create readable mappings to its own page frames, regardless of their current types. A frame may only safely be retasked when its reference count is zero. This mechanism is used to maintain the invariants required for safety; for example, a domain cannot have a writable mapping to any part of a page table as this would require the frame concerned to simultaneously be of types PT and RW.

3. Two Awesome Vulnerabilities

3.1. XSA-148(Dome Breaking)

The vulnerability of XSA-148 was named as Dome Breaking by our team. This vulnerability was titled as *x86: Uncontrolled creation of large page mappings* by PV guests by Xen Project Team and its official description is:

> The code to validate level 2 page table entries is bypassed when certain conditions are satisfied. This means that a PV guest can create writeable mappings using super page mappings. Such writeable mappings can violate Xen intended invariants for pages which Xen is supposed to keep read-only. This is possible even if the ällowsuperpagečommand line option is not used.

When PV Guest update their level 2 page tables(aka. L2T or PDT), the mod_l2_entry() function is responsible to validate the new level 2 page table entries(aka. L2E, L2TE, PDE or PDTE). Code 1 give a simplified version of the mod_l2_entry() function.

```
/* Xen 4.6 xen/x86/mm.c */
/* Update the L2 entry at pl2e to new value nl2e. pl2e is within frame pfn. */
static int mod_l2_entry(l2_pgentry_t *pl2e,
                        l2_pgentry_t nl2e,
                        unsigned long pfn,
                        int preserve_ad,
                        struct vcpu *vcpu)
{
    l2_pgentry_t ol2e;
    struct domain *d = vcpu->domain;
    struct page_info *l2pg = mfn_to_page(pfn);
    unsigned long type = l2pg->u.inuse.type_info;
    int rc = 0;

    if (unlikely(!is_guest_l2_slot(d, type, pgentry_ptr_to_slot(pl2e))))
    {
    // skip ...
    }

1829     if ( unlikely(__copy_from_user(&ol2e, pl2e, sizeof(ol2e)) != 0) )
1830         return -EFAULT;
1831
1832     if ( l2e_get_flags(nl2e) & _PAGE_PRESENT )
1833     {
1834         if ( unlikely(l2e_get_flags(nl2e) & L2_DISALLOW_MASK) )
1835         {
               // skip ...
1839         }
1840
1841         /* Fast path for identical mapping and presence. */
1842         if ( !l2e_has_changed(ol2e, nl2e, _PAGE_PRESENT) )
1843         {
1844             adjust_guest_l2e(nl2e, d);
1845             if (UPDATE_ENTRY(l2, pl2e, ol2e, nl2e, pfn, vcpu, preserve_ad))
1846                 return 0;
1847             return -EBUSY;
1848         }
1849
1850         if ( unlikely((rc = get_page_from_l2e(nl2e, pfn, d)) < 0) )
1851             return rc;
1852
1853         adjust_guest_l2e(nl2e, d);
1854         if ( unlikely(!UPDATE_ENTRY(l2, pl2e, ol2e, nl2e, pfn, vcpu,
1855                                     preserve_ad)) )
1856         {
1857             ol2e = nl2e;
1858             rc = -EBUSY;
1859         }
1860     }
1861     else if ( unlikely(!UPDATE_ENTRY(l2, pl2e, ol2e, nl2e, pfn, vcpu,
1862                                      preserve_ad)) )
1863     {
1864         return -EBUSY;
1865     }
1866
1867     put_page_from_l2e(ol2e, pfn);
1868     return rc;
1869 }
```

Code 1. mod_l2_entry() function

In the Code 1, validation at line 1823 guarantees that the target L2T belongs to the current PV Guest. The current PV Guest is the one who are requesting for the L2T update. Validation at line 1832 check whether the P flag of new L2TE is set or not. If not, the mod_l2_entry() function considers its safe to update L2T with the new L2TE immediately. But if the P flag exists, more validations should to be performed. At line 1834, flags defined by micro L2_DISALLOW_MASK should be cleared in the new L2TE. These flags dont include _PAGE_PRESENT, _PAGE_RW, _PAGE_USER, _PAGE_ACCESSED, _PAGE_DIRTY, _PAGE_AVAIL and _PAGE_PSE. At line 1842, the fast-update-path designed for performance improvement allows this update if MFN and P bits of new entry are identical to those of the old one. At line 1850, the get_page_from_l2e() function will determine whether the update request should be applied if the fast-update-path rejected to executed. The get_page_from_l2e() function will perform other validations and we will analyze it at next section. Now we will describe the weakness introduced by fast-update-path.

As described above, fast-update-path only checks two fields of new L2TE: MFN and P flag. If a PV Guest gives a new L2TE satisfied with two rules:

1. new L2TEs l2_disallowed_flags == 0
2. new L2TE.P == old L2TE.P == 1
3. new L2TE.MFN == old L2TE.MFN

The fast-update-path would work and update operation would be applied. Based on the above L2TE, append two another rules as follow:

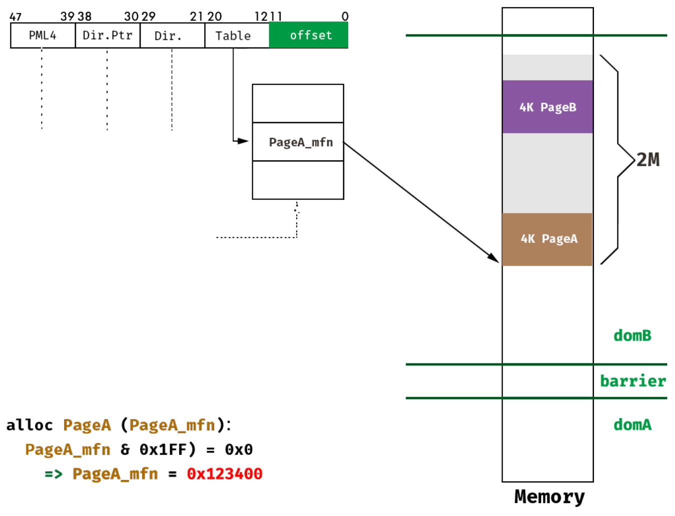
4. new L2TE.PSE == 1
5. new L2TE.W == 1

The enabled PSE flag let MMU work under Intel IA-32e 2M paging mode while W flag allows the write access.

Because constraint **4** and **5** satisfy the fast-update-path validation, the new L2TE also would be accepted. Then, we will immediately get a writable 2M memory bypass the superpage validation that should be performed in the get_page_from_l2e() function. If we have put a L1 page table in the 2M memory area before the malicious L2T update, the L1 page table could be writable directly. We could create writable mappings to any machine frame by directly modifying the writable L1 page table without any hypercall requests or validations and all safe invariants of memory access would be break out.

For clearly, important steps are described as follows:

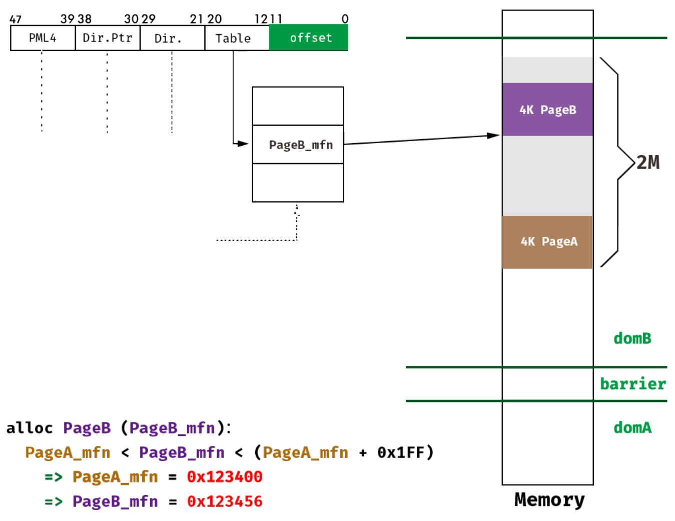
1. Allocate PageA:



```
alloc PageA (PageA_mfn):
  PageA_mfn & 0x1FF) = 0x0
    => PageA_mfn = 0x123400
```

FIGURE 2. step1

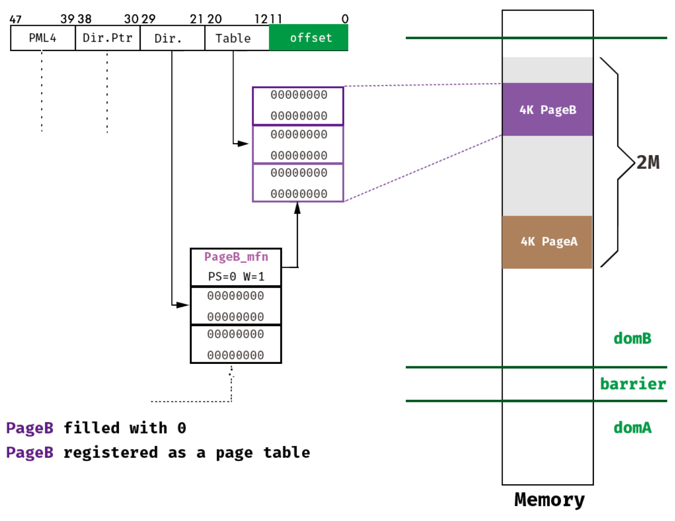
2. Allocate PageB:



```
alloc PageB (PageB_mfn):
  PageA_mfn < PageB_mfn < (PageA_mfn + 0x1FF)
    => PageA_mfn = 0x123400
    => PageB_mfn = 0x123456
```

FIGURE 3. step2

3. Register empty PageB as a page table:



```
PageB filled with 0
PageB registered as a page table
```

FIGURE 4. step3

4. Register empty PageA as a page table:

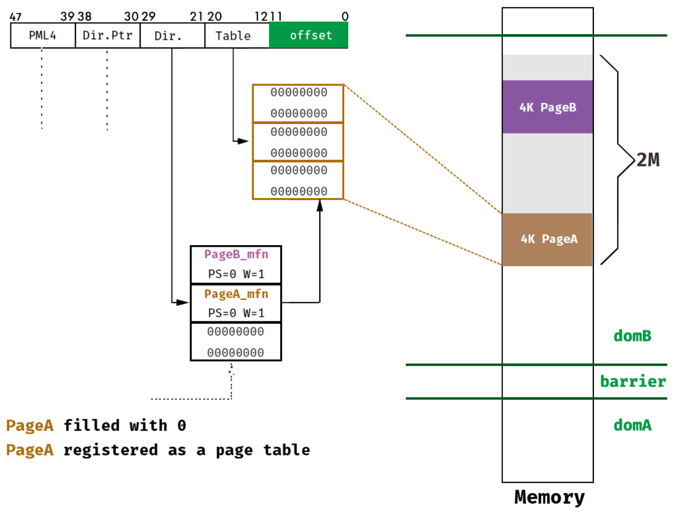


FIGURE 5. step4

5. Enable PSE flag to gain writable access of PageB:

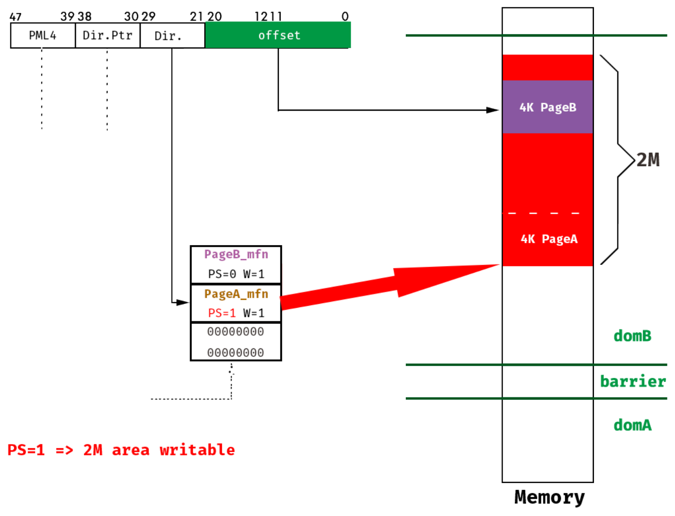


FIGURE 6. step5

6. Modify PageB directly to access whole memory region:

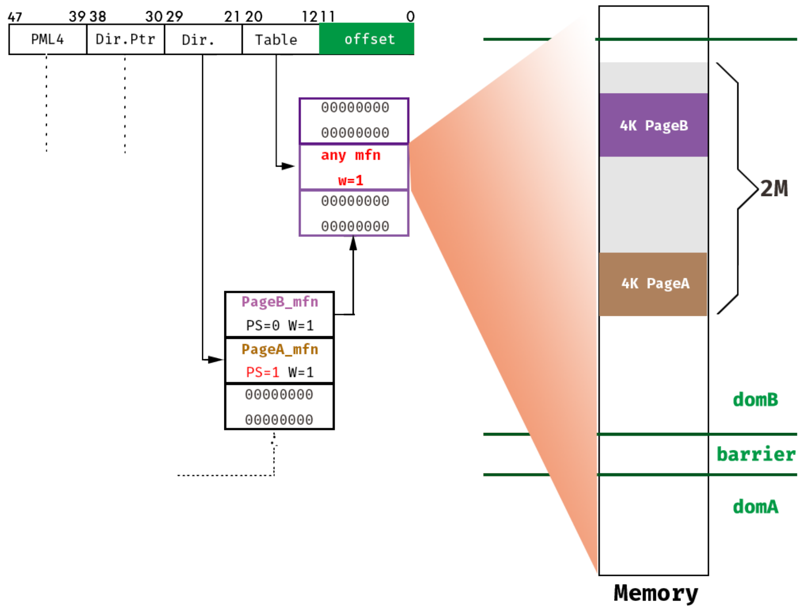


FIGURE 7. step6

Since all machine memory could be read and write arbitrarily, the VM escape is on the way. Some exploitation technologies for VM escape will be given at section 4.

3.2. XSA-182(Ouroboros)

Inspired by Ouroboros, an ancient symbol with a snake biting its tail, we found another critical vulnerability in the memory management logic. The new vulnerability allow a level 2 page table reference itself as a level 1 page table, or a level 3 page table reference itself as a level 2 page table, or a level 4 page table reference itself as a level 3 page table while all of these page table self-reference are assigned with W flags. For simplicity, I will only discuss the circumstance of level 2 page table self-reference with W flag. Others are similar to this.

At section 3.1, we have mentioned that the mod_l2_entry() function would call get_page_from_l2e() function to perform more validations if fast-update-path check failed. Code 2 describes the implementation of get_page_from_l2e() function.

```
/* Xen 4.6 xen/x86/mm.c */
/* NB. Virtual address 'l2e' maps to a machine address within frame 'pfn'. */
define_get_linear_pagetable(l2);
static int
get_page_from_l2e(
    l2_pgentry_t l2e, unsigned long pfn, struct domain *d)
{
  unsigned long mfn = l2e_get_pfn(l2e);
  int rc;

  if ( !(l2e_get_flags(l2e) & _PAGE_PRESENT) )
    return 1;
959
960    if ( unlikely((l2e_get_flags(l2e) & L2_DISALLOW_MASK)) )
961    {
962      MEM_LOG("Bad␣L2␣flags␣%x", l2e_get_flags(l2e) & L2_DISALLOW_MASK);
963      return -EINVAL;
964    }
965
966    if ( !(l2e_get_flags(l2e) & _PAGE_PSE) )
967    {
968      rc = get_page_and_type_from_pagenr(mfn, PGT_l1_page_table, d, 0, 0);
969      if ( unlikely(rc == -EINVAL) && get_l2_linear_pagetable(l2e, pfn, d) )
970        rc = 0;
971      return rc;
972    }
973
974    if ( !opt_allow_superpage )
975    {
976      MEM_LOG("Attempt␣to␣map␣superpage␣without␣allowsuperpage␣"
977              "flag␣in␣hypervisor");
978      return -EINVAL;
979    }
980
981    if ( mfn & (L1_PAGETABLE_ENTRIES-1) )
982    {
983      MEM_LOG("Unaligned␣superpage␣map␣attempt␣mfn␣%lx", mfn);
984      return -EINVAL;
985    }
986
987    return get_superpage(mfn, d);
988 }
```

Code 2. get_page_from_l2e() function

The code block from line 966 will verify whether the page type of target page is valid or not via get_page_and_type_from_pagenr() function. The target page is indicated by the new L2TEs MFN and the valid page type should be PGT_l1_page_table at here. That means a level 2 table page entry should refer to a level 1 page table. At most situations, the get_page_and_type_from_pagenr() function would return 0 means the target page has been assigned a correct page type. If the page type of given page isnt consistent with the required one, the get_page_and_type_from_pagenr() function return EINVAL and the get_l2_linear_pagetable() function, actually defined by a micro, will get involved. The get_l2_linear_pagetable() function is defined as Code 3.

```
/* Xen 4.6 xen/x86/mm.c */
/*
 * We allow root tables to map each other (a.k.a. linear page tables). It
 * needs some special care with reference counts and access permissions:
 *  1. The mapping entry must be read-only, or the guest may get write access
 *     to its own PTEs.
 *  2. We must only bump the reference counts for an *already validated*
 *     L2 table, or we can end up in a deadlock in get_page_type() by waiting
 *     on a validation that is required to complete that validation.
 *  3. We only need to increment the reference counts for the mapped page
 *      frame if it is mapped by a different root table. This is sufficient and
 *      also necessary to allow validation of a root table mapping itself.
 */
660 #define define_get_linear_pagetable(level)                                  \
661 static int                                                                  \
662 get_##level##_linear_pagetable(                                             \
663     level##_pgentry_t pde, unsigned long pde_pfn, struct domain *d)         \
664 {                                                                           \
665     unsigned long x, y;                                                     \
666     struct page_info *page;                                                 \
667     unsigned long pfn;                                                      \
668                                                                             \
669     if ( (level##e_get_flags(pde) & _PAGE_RW) )                             \
670     {                                                                       \
671         MEM_LOG("Attempt to create linear p.t. with write perms");          \
672         return 0;                                                           \
673     }                                                                       \
674                                                                             \
675     if ( (pfn = level##e_get_pfn(pde)) != pde_pfn )                         \
676     {                                                                       \
        // return 0 if validations failed                                       \
698     }                                                                       \
699                                                                             \
700     return 1;                                                               \
701 }
```

Code 3. get_l2_linear_pagetable() function

At line 669 of the Code 3, the new L2TE with W flag will cause this function return 0 which means validation failed. For a L2TE without W flag, its MFN field will be compared to the MFN of current level 2 page table. If equals, some validations will be ignored and this function return SUCCESS! In actually, this odd codes and logics are designed for recursive mappings. At here, the memory management allows a level 2 page table has itself references and, of cause, line 669 of the get_l2_linear_pagetable() function only accepts read-only self-references.

Back to the fast-update-path of mod_l2_entry() function at now. As is mentioned at section 3.1, fast-update-path allows to update target L2T immediately for identical mappings. If we have has a level 2 read-only self-reference, then we could transform the self-reference from read-only to read-write. Consider seriously about the current level 2 page table references. L2T has an entry with W flag referring to this L2T itself. It means this L2T will be treated as a normal level 1 page table by MMU and the level last page table has a writable entry referring to the L2T. This recursive mapping cause the malicious PV Guest could write its level 2 page table directly. Similar to what we discussed at section 3.1, safe invariants of memory access would be broke out again and a malicious could arbitrarily read and write the whole machine memory.

For clearly, important steps are described as follows:

1. Allocate PageA and registered is as a PDT:

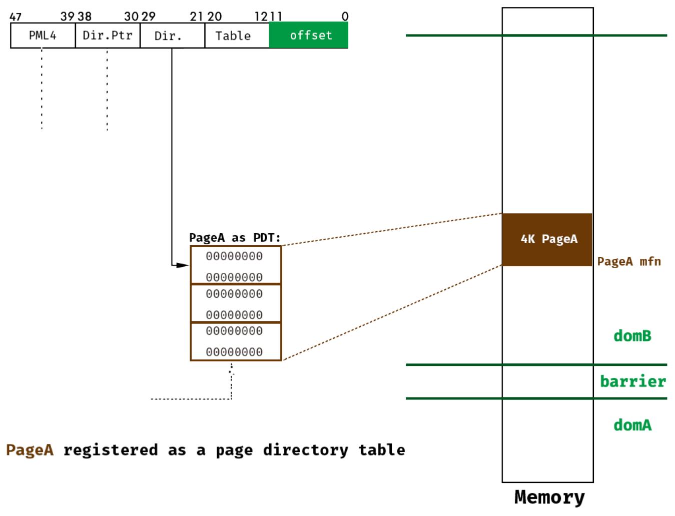


FIGURE 8. step1

2. Modify PageA to let PageA[0] reference itself:

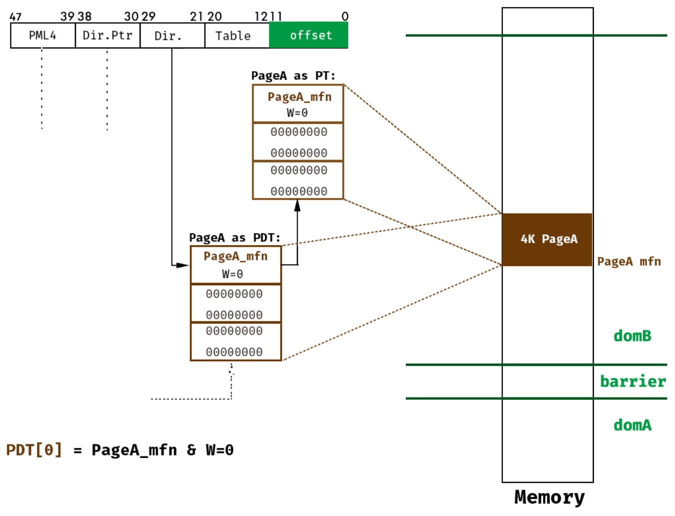


FIGURE 9. step2

3. Now PageA could be read-only accessed by PDT and PT PageA:

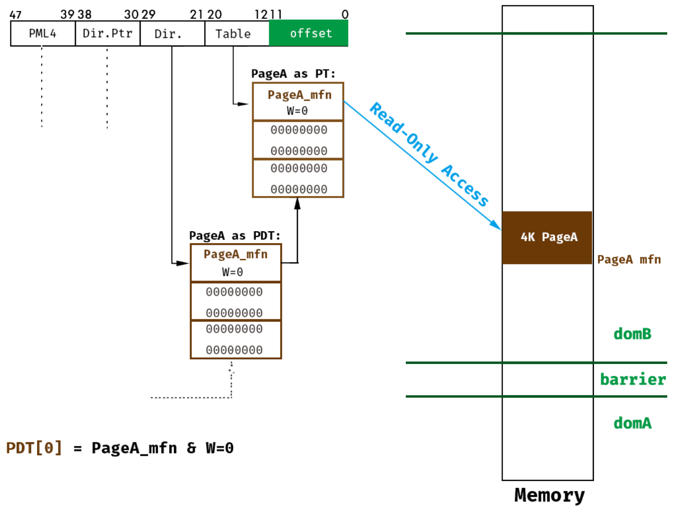


FIGURE 10. step3

4. Change PDT[1].W to 1 via fast-path update:

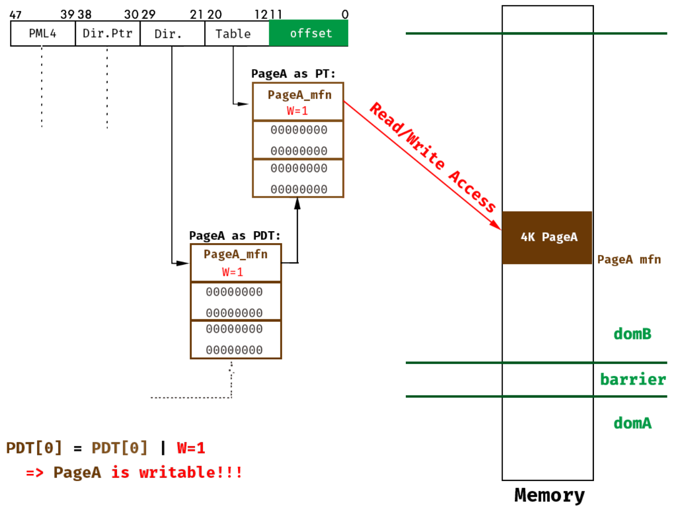


FIGURE 11. step4

5. Add a writable PT PageB via direct PageA modification:

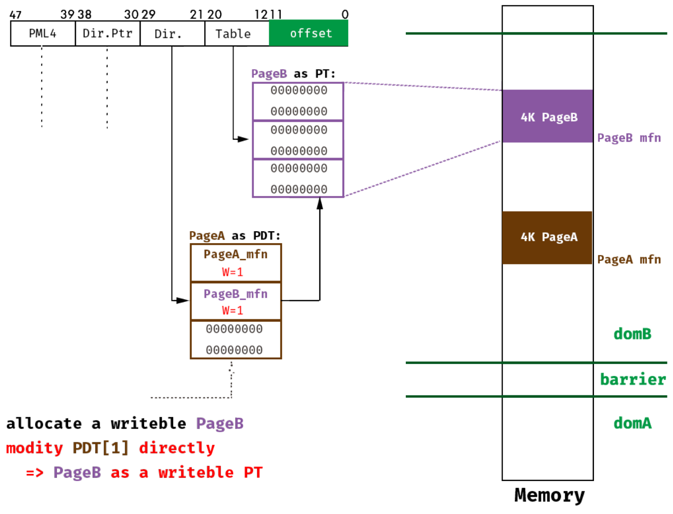


FIGURE 12. step5

6. Access whole memory via direct PT PageB Modification

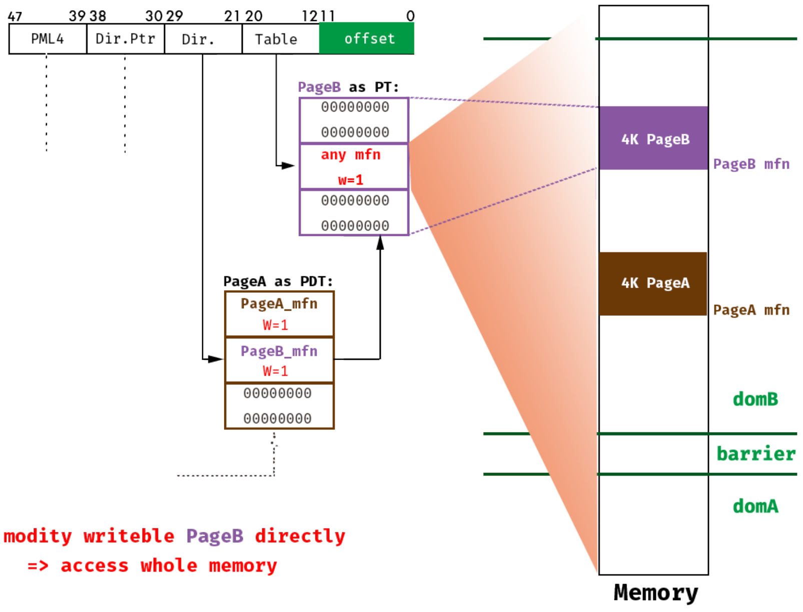


FIGURE 13. step6

4. Exploitation Technologies

Since we have been able to read and write arbitrary machine memory, control any other domains running on the same physical machine isnt difficult. Similar to traditional exploitation technologies at system level, the fundamental principle is special memory search and modification.

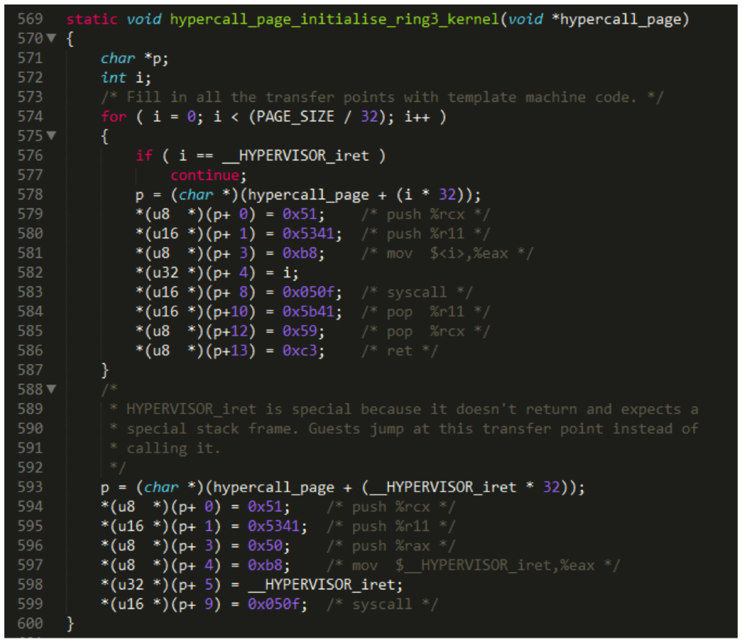
At follow sections, some lethal exploitation technologies will be discussed.

4.1. Special Page Hijack

Under the virtualization environments, guest domains know nothing about other domains.

A special page refers to a page filled with special contents. Thus its certainly possible to locate these pages via signatures of their contents. If a special page was filled with instructions that would be executed by the target domain and has enough spare space to be used to deploy our malicious payloads, the control flow of target domain would be easily hijacked. Typically, VSyscall page or VDSO page in Linux based system or SharedUserData page in Windows are ideal target for special page hijack attack. In this paper, we will discuss another special page introduced by Xen and describe exploitation technologies based on it.

In order to facilitate guest domains to request hypercalls, Xen Hypervisor help every PV guest domain to initialize a Hypercall Page for it. This page is in guest domain context but its content is provided by the hypervisor, transparently to the guest. The content just is hypercall stub codes which will be executed when guest domain kernel make hypercall requests.



```c
static void hypercall_page_initialise_ring3_kernel(void *hypercall_page)
{
    char *p;
    int i;
    /* Fill in all the transfer points with template machine code. */
    for ( i = 0; i < (PAGE_SIZE / 32); i++ )
    {
        if ( i == __HYPERVISOR_iret )
            continue;
        p = (char *)(hypercall_page + (i * 32));
        *(u8  *)(p+ 0) = 0x51;    /* push %rcx */
        *(u16 *)(p+ 1) = 0x5341;  /* push %r11 */
        *(u8  *)(p+ 3) = 0xb8;    /* mov  $<i>,%eax */
        *(u32 *)(p+ 4) = i;
        *(u16 *)(p+ 8) = 0x050f;  /* syscall */
        *(u16 *)(p+10) = 0x5b41;  /* pop  %r11 */
        *(u8  *)(p+12) = 0x59;    /* pop  %rcx */
        *(u8  *)(p+13) = 0xc3;    /* ret */
    }
    /*
     * HYPERVISOR_iret is special because it doesn't return and expects a
     * special stack frame. Guests jump at this transfer point instead of
     * calling it.
     */
    p = (char *)(hypercall_page + (__HYPERVISOR_iret * 32));
    *(u8  *)(p+ 0) = 0x51;    /* push %rcx */
    *(u16 *)(p+ 1) = 0x5341;  /* push %r11 */
    *(u8  *)(p+ 3) = 0x50;    /* push %rax */
    *(u8  *)(p+ 4) = 0xb8;    /* mov  $__HYPERVISOR_iret,%eax */
    *(u32 *)(p+ 5) = __HYPERVISOR_iret;
    *(u16 *)(p+ 9) = 0x050f;  /* syscall */
}
```

FIGURE 14. Hypercall Page Initialization

The page is 4096B while every hypercall stub codes is 32B in size. Bacause the hypervisor has at most 64 hypercalls at now, the second half of this Hypercall Page is never used. Every PV guest domain only hold one Hypercall Page and HVM guest domain would also hold one if it install a PV driver. Thus, Hypercall Page is an wonderful target for hijack if we want to execute payloads with OS kernel privilege in guest domains context. For example, if we want to get a root shell of dom0, exploitation steps should be taken as follow:

1. Search Hypercall Page signature from memory beginning. The first match must belong to dom0. This step maybe spent 10 to 30 minutes depended on specific environments. According to the content of Hypercall Page, its signature should be:
   (hex bytes) 51 41 53 B8 00 00 00 00 0F 05 41 5B 59 C3 CC CC

2. Deploy malicious payloads or shellcodes at the second half of the Hypercall Page founded at step 1. The memory of 2048 bytes is quite enough for the a basic implementation of a shell program with a ring buffer.

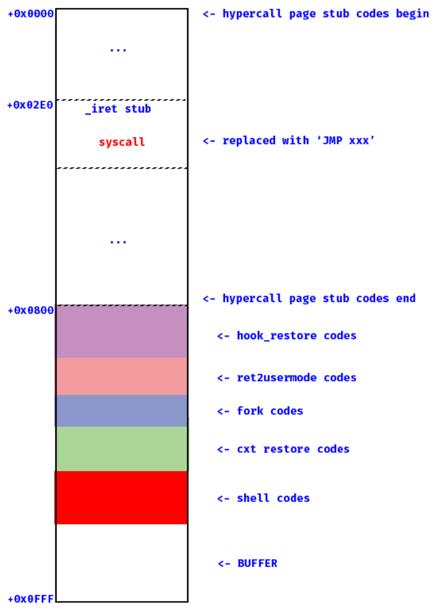


FIGURE 15. Hypercall Page With Malicious Payloads

3. Modify hypercall stub codes to let control flow transfer to malicious payloads. Then we could execute arbitrary commands with root privilege in dom0 context.

4.2. VMM Code Injection

This section will disscuss how to execute malicious shellcodes in VMM context.

Just like syscall table in Linux kernel, there is a hypercall table in Xen Hypervisor. The hypercall table holds function pointers of hyercalls. It has 64 slots but only a half are in use. We could modify this table and add custom hypercalls dynamicly.

```
/* Xen 4.6 xen/x86/x6_64/entry.S */
ENTRY(hypercall_table)
        .quad do_set_trap_table     /* 0 */
        .quad do_mmu_update
        .quad do_set_gdt
        .quad do_stack_switch
        .quad do_set_callbacks
        .quad  do_fpu_taskswitch      /*  5 */
   // skip ...
        .quad  do_ni_hypercall        /* reserved for XenClient */
        .quad  do_xenpmu_op           /* 40 */
        .rept  __HYPERVISOR_arch_0-((.-hypercall_table)/8)
        .quad  do_ni_hypercall
        .endr
        .quad  do_mca                 /* 48 */
        .quad  paging_domctl_continuation
        .rept  NR_hypercalls-((.-hypercall_table)/8)
        .quad  do_ni_hypercall
        .endr

ENTRY(hypercall_args_table)
        .byte 1 /* do_set_trap_table      */  /*  0 */
        .byte 4 /* do_mmu_update          */
        .byte 2 /* do_set_gdt             */
        .byte 2 /* do_stack_switch        */
        .byte 3 /* do_set_callbacks       */
        .byte 1 /* do_fpu_taskswitch      */  /*  5 */
        .byte 2 /* do_sched_op_compat     */
        .byte 1 /* do_platform_op         */
        .byte 2 /* do_set_debugreg        */
        .byte 1 /* do_get_debugreg        */
        .byte 2 /* do_update_descriptor   */  /* 10 */
   // skip ...
        .rept  __HYPERVISOR_arch_0-(.-hypercall_args_table)
        .byte 0 /* do_ni_hypercall        */
        .endr
        .byte 1 /* do_mca                 */  /* 48 */
        .byte 1 /* paging_domctl_continuation */
        .rept  NR_hypercalls-(.-hypercall_args_table)
        .byte 0 /* do_ni_hypercall        */
        .endr
```

Code 4. hypercall_table and hypercall_args_table

Malicious guest domain has to locate this table via memory search since we know nothing about it. Although the hypercall table holds function pointers and doesnt have a well-marked signature, we also could gain its location with the aid of another well-marked page, the hypercall args table. The hypercall args table only stores arguments numbers for all hypercalls and its content is fixed. The hypercall args table is contiguous to the hypercall table at the view of page frame number. If mfn of hypercall args table we have gained equals to n, n-1 is the target mfn of hypercall table we needed.

Besides the hypercall table location, another problem is how to map malicious shellcodes into hypervisors memory space. Fortunately, we needn't to gain and modify hypervisor's page tables. At the point of MMU, memory space of the guest domain is consistent with the one of hypervisor. This is very like that the memory space of usermode process is visible by kernel in Linux. So as a malicious guest domain, we just need allocate a malicious page and deploy our shellcode in it. Then modify attributes of corresponding page table entry to allow this page to be executable in privileged mode. While PV Guest request the fake hypercall, malicious codes will be executed within hypervisor context.

For example, if our unprivileged guest domain want to get all privileges, the is_privileged field of its domain struct should be set.

1. allocate a memory area and deploy code in it:

```
current->domain->is_privileged=1:
    mov  $0xffffffffffff8000,%rax
    and  %rsp,%rax
    mov  0x7fe8(%rax),%rax
    mov  0x10(%rax),%rax
    movb $0x1,0x116(%rax)
    retq
```

2. search hypercall_table and modify slot N to refer to the memory area

3. bypass SMEP and SMAP features: responded PTE.U/S = 0

4. request this hypercall:

```
MOV N,%RAX / SYSCALL
```

5. Conclusions

Vulnerabilities like XSA-148 and XSA-182 are extremely rare. At the point of seriousness and exploitation, they should belong to the top-class group and beyond over all others listed in Xen Security Advisory board. VM escape is no longer just a unreal legend and there are more things need to be explored. We expect our work in this paper could provide helpful guidelines for future research on virtualization security.

6. Acknowledgements

I would like to thank my original team leaders, flashsky and _alert7, for instructing me to start my research on virtualization security. And Im grateful to my colleagues, especially Donghai Zhu and Zuozhi Fan, for their help on the verification of all kinds of ideas. This work is supported by the Alibaba Cloud. Any views or opinions presented are solely those of the author and do not necessarily represent those of Alibaba Cloud.

References

1. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauery, Ian Pratt, Andrew Wareld, “Xen and the Art of Virtualization”, 2003
2. Xen Project Security Team, “XSA-148/CVE-2015-7835”, 2015, https://xenbits.xen.org/xsa/advisory-148.html
3. Xen Project Security Team, “XSA-182/CVE-2016-6258”, 2016, https://xenbits.xen.org/xsa/advisory-182.html
4. Xen Project Team, “Offical Documents on Xen”, http://wiki.xen.org/
5. Intel Inc., “Intel 64 and IA-32 Architectures Software Developers Manual”, Vol. 3.