Technical Report: SPEC-RG-2014-001
Version: 1.0

Technical Information on Vulnerabilities of Hypercall Handlers

SPEC RG IDS Benchmarking Working Group

Aleksandar Milenkoski
Institute for Program Structures and Data Organization
Karlsruhe Institute of Technology
Karlsruhe, Germany
milenkoski@kit.edu

Marco Vieira
CISUC, Department of Informatics Engineering
University of Coimbra
Coimbra, Portugal
mvieira@dei.uc.pt

Bryan D. Payne
Director, Security Research
Nebula Inc.
Mountain View, CA, USA
bdpayne@acm.org

Nuno Antunes
CISUC, Department of Informatics Engineering
University of Coimbra
Coimbra, Portugal
nmsa@dei.uc.pt

Samuel Kounev
Chair of Software Engineering
University of Würzburg
Würzburg, Germany
samuel.kounev@uni-wuerzburg.de

Contents

Executive Summary

Modern virtualized service infrastructures expose attack vectors that enable attacks of high severity, such as attacks targeting hypervisors. A malicious user of a guest VM (virtual machine) may execute an attack against the underlying hypervisor via hypercalls, which are software traps from a kernel of a fully or partially paravirtualized guest VM to the hypervisor. The exploitation of a vulnerability of a hypercall handler may have severe consequences such as altering hypervisor's memory, which may result in the execution of malicious code with hypervisor privilege. Despite the importance of vulnerabilities of hypercall handlers, there is not much publicly available information on them. This significantly hinders advances towards securing hypercall interfaces. In this work, we provide in-depth technical information on publicly disclosed vulnerabilities of hypercall handlers. Our vulnerability analysis is based on reverse engineering the released patches fixing the considered vulnerabilities. For each analyzed vulnerability, we provide background information essential for understanding the vulnerability, and information on the vulnerable hypercall handler and the error causing the vulnerability. We also show how the vulnerability can be triggered and discuss the state of the targeted hypervisor after the vulnerability has been triggered.

Keywords:[1]
Security and Privacy - Systems security - Operating systems security - Virtualization and security

[1]The keywords used here are defined as part of The 2012 ACM Computing Classification System [19].

1 Introduction

Virtualized environments are becoming increasingly ubiquitous with the growing proliferation of virtualized data centers and cloud environments. However, security concerns are still one of the greatest showstoppers for the wide adoption of cloud computing [23]. Attackers are actively exploring virtualization-specific attack surfaces, such as hypervisors. Attacks targeting hypervisors are of high severity since they may result in crashing the hypervisors including all guest VMs (virtual machines) running on top of them or in altering hypervisors' memory.

A malicious guest VM user may execute an attack against the underlying hypervisor via hypercalls, which are software traps from a kernel of a fully or partially paravirtualized guest VM to the hypervisor. Hypercalls enable intrusion into vulnerable hypervisors initiated from a malicious guest VM kernel. As Rutkowska et al. [25] demonstrate, the exploitation of a vulnerability of a hypercall handler (i.e., a hypercall vulnerability) may lead to altering the memory of the targeted hypervisor, which enables, for example, the execution of malicious code with hypervisor privilege.

Given the severity of attacks triggering hypercall vulnerabilities, the characterization of the hypercall attack surface is a priority since it is crucial for better understanding the security threats posed by hypercall interfaces. The lack of such an understanding significantly hinders advances towards monitoring and securing these interfaces. In-depth technical information on hypercall vulnerabilities is a requirement for characterizing the hypercall attack surface. However, such information is currently very limited. Publicly disclosed vulnerability reports describing hypercall vulnerabilities (e.g., CVE-2013-4494, CVE-2013-3898) are typically the sole source of information and provide only high-level descriptions. There is also no publicly available information on attacks triggering hypercall vulnerabilities performed in practice.

The goal of this work is to provide technical information on hypercall vulnerabilities needed for the improvement of the security of hypercall interfaces (e.g., information on the errors that caused the vulnerabilities and how the vulnerabilities can be triggered). To this end, we analyzed all publicly disclosed hypercall vulnerabilities that we found by searching major CVE (Common Vulnerability and Exposures) report databases (e.g., cvedetails [22]) based on relevant keywords, such as names of operations of hypercalls. In this work, we focus on the vulnerabilities described in the vulnerability reports CVE-2012-3494, CVE-2012-3495, CVE-2012-3496, CVE-2012-4539, CVE-2012-5510, CVE-2012-5513, CVE-2012-5525, and CVE-2013-1964. These vulnerabilities are representative of the vulnerabilities that we analyzed in terms of the errors causing them and the ways in which they can be triggered. The vulnerabilities considered in this work are from the Xen hypervisor [20], which has the most extensive hypercall interface as opposed to other hypervisors, such as KVM [24]. The considered vulnerabilities are in the handlers of the hypercalls *memory_op*, *gnttab_op*, *set_debugreg*, *physdev_op*, and *mmuext_op*.

Our approach for analyzing a hypercall vulnerability consisted of the following steps: *(i)* analysis of the CVE report describing the vulnerability and other relevant information sources, for example, security advisories; *(ii)* reverse engineering of the released patch fixing the vulnerability, and *(iii)* developing proof-of-concept code for triggering the vulnerability. For each considered vulnerability, we provide background information essential for understanding the vulnerability, and information on the vulnerable hypercall handler (i.e., information about the workflow, and input and output data of the handler) and the error causing the vulnerability. We also show how the vulnerability can be triggered and discuss the state of the targeted hypervisor after the vulnerability has been triggered.

We stress that we provide information on a vulnerable hypercall handler to the extent that is relevant for understanding a given vulnerability, for example, we discuss only some input parameters of the handler. We also stress that we do not provide proof-of-concept code for triggering the considered vulnerabilities ready for use. We present only the hypercalls executed

as part of an attack triggering a given hypercall vulnerability, and the values of relevant hypercall parameters (i.e., parameters identifying the executed hypercalls and, where applicable, parameters with values specifically crafted for triggering the vulnerability). Finally, we stress that we do not demonstrate vulnerability exploitation where it is possible (e.g., malicious code execution). We focus instead on the errors causing the considered vulnerabilities, the activities for triggering them, and the effects of triggering the vulnerabilities on the state of the vulnerable hypervisors. We argue that the information that we provide is relevant for better understanding the security threats that hypercall interfaces pose, which will help to focus approaches for improving the security of hypervisors.

2 Information on hypercall vulnerabilities

2.1 Hypercall memory_op

The *memory_op* hypercall is used for managing the memory of a guest VM, for example, altering the layout of a given memory region.[1] In the handler of *memory_op*, the different types of memory addresses that the Xen hypervisor supports for abstracting physical memory available to guest VMs are used for accessing locations in memory:

- *virtual address* - an address of a location in the virtual memory of a guest VM;
- *GPFN (Guest Pseudo-Physical Frame Number)* - an address of a page frame that is a physical memory address from the perspective of a guest VM;
- *GMFN (Guest Machine Frame Number)* - an address of a page frame that is a machine address from the perspective of a guest VM;
- *MFN (Machine Frame Number)* - an address of a page frame that is a real machine address.

For accessing contiguous memory blocks, the different types of addresses mentioned above are used for accessing *extents* of a given *order* such that an extent consists of 2^{order} memory pages.

Mappings between the different types of memory addresses are stored in tables for that purpose. Mappings between virtual addresses and GPFNs are stored in a page table, between GPFNs and GMFNs in a physical-to-machine table, and between GMFNs and GPFNs in a machine-to-physical table.

We refer to reader to [18] and [21] for further information on how the Xen hypervisor manages memory.

Vulnerability CVE-2012-3496

> "XENMEM_populate_physmap in Xen 4.0, 4.1, and 4.2, and Citrix XenServer 6.0.2 and earlier, when translating paging mode is not used, allows local PV OS guest kernels to cause a denial of service (BUG triggered and host crash) via invalid flags such as MEMF_populate_on_demand." [3]

XENMEM_populate_physmap is an operation of the *memory_op* hypercall, which is used for requesting extents from the hypervisor. *XENMEM_populate_physmap* is also used for marking extents as "populate-on-demand". Extents marked as "populate-on-demand" can be assigned to the physical memory of a given guest VM, or removed from it, on demand at run time.

Input:[2] *XENMEM_populate_physmap* takes as input a structure of type *xen_memory_reservation.* which is defined as:

[1] We refer the reader to [9] for more information on the functionalities of the *memory_op* hypercall.
[2] As in Xen of version 4.1.0.

```
struct xen_memory_reservation {
  GUEST_HANDLE(xen_pfn_t) extent_start;
  unsigned int  extent_order;
  unsigned int  address_bits;
  . . .
}
```

extent_start stores the virtual address of the head of an array that contains memory addresses (GPFNs) at which the extents obtained from the hypervisor are to be mapped, or addresses (GPFNs) of the beginnings of the extents that are to be marked as "populate-on-demand"; *extent_order* stores the order of a single extent; *address_bits* stores the flags of the *XENMEM_populate_physmap* hypercall operation, one of which is *MEMF_populate_on_demand*. *MEMF_populate_on_demand* is enabled when *XENMEM_populate_physmap* is used for marking extents as "populate-on-demand".

Output:[2] On success, *XENMEM_populate_physmap* returns the number of the obtained extents or of the extents marked as "populate-on-demand". In case *XENMEM_populate_physmap* has been used for obtaining extents, the array that starts at the virtual address stored in *extent_start* is populated with the memory addresses (MFNs) of the beginnings of the obtained extents. On failure, *XENMEM_populate_physmap* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[2]

```
do_memory_op (XENMEM_populate_physmap, (struct xen_memory_reservation) res)
  . . .
  call populate_physmap(...)
    . . .
      for each GPFN in res.extent_start
        if MEMF_populate_on_demand
          call guest_physmap_mark_populate_on_demand(...)
            call BUG_ON(...)
            . . .
          return
          . . .
        else:
          . . .
    . . .
  return
return
```

Description of the vulnerability: In *guest_physmap_mark_populate_on_demand*, a function invoked in the handler of *XENMEM_populate_physmap*, the BUG_ON macro is used for checking whether the guest VM from where the *memory_op* hypercall has been invoked has the "translated paging" mode disabled. BUG_ON is a macro that crashes the system where it is executed if the condition that it evaluates is true. If *guest_physmap_mark_populate_on_demand* is invoked from a paravirtualized guest VM (note that paravirtualized guest VMs have the "translated paging" mode disabled by default), the condition that the BUG_ON macro evaluates is true and the hypervisor crashes. Thus, CVE-2012-3496 can be triggered by invoking the *XENMEM_populate_physmap* hypercall operation, with the *MEMF_populate_on_demand* flag enabled, from a paravirtualized guest VM.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-3496 was released on 5 September 2012 and is available at [12]. The patch replaces the *BUG_ON* macro with an *if* clause.

Triggering CVE-2012-3496: We triggered CVE-2012-3496 in the following environment:

- guest VM: OS - Debian Squeeze (64 bit), kernel - 2.6.32-5-amd64;
- host VM: OS - Debian Squeeze (64 bit), kernel - 2.6.32-5-amd64;
- hypervisor: Xen 4.1.0.

The attack that we executed is depicted in Figure 2.1.

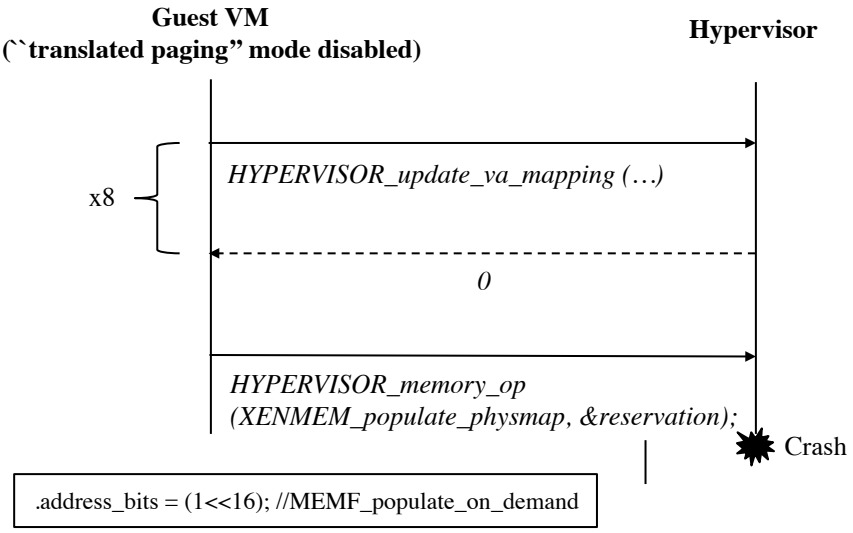

Figure 2.1: An attack triggering CVE-2012-3496

Post-attack state of the hypervisor: The hypervisor crashes when the BUG_ON macro is executed.

Vulnerability CVE-2012-5513

> "The XENMEM exchange handler in Xen 4.2 and earlier does not properly check the memory address, which allows local PV guest OS administrators to cause a denial of service (crash) or possibly gain privileges via unspecified vectors that overwrite memory in the hypervisor reserved range." [6]

XENMEM_exchange is an operation of the *memory_op* hypercall, which is used for modifying the layout of a memory region of a guest VM by "exchanging" extents between the guest VM and the hypervisor. The latter is performed by remapping a set of memory addresses (GPFNs) of beginnings of extents of the guest VM to memory addresses (GMFNs) of beginnings of extents, requested by the guest VM and allocated by the hypervisor for the "exchange" operation. For instance, *XENMEM_exchange* can be used for defragmenting memory such that, for example, 2 extents consisting of 2 pages are exchanged for a single extent consisting of 4 pages.

Input:[3] *XENMEM_exchange* takes as input a structure of type *xen_memory_exchange* defined as:

```
struct xen_memory_exchange {
  struct xen_memory_reservation in;
  struct xen_memory_reservation out;
  xen_ulong_t nr_exchanged;
}
```

, where *xen_memory_reservation* is defined as:

[3] As in Xen of version 4.1.0.

```
struct xen_memory_reservation {
  GUEST_HANDLE(xen_pfn_t) extent_start;
  unsigned int extent_order;
  xen_ulong_t nr_extents;
  . . .
}
```

The fields of the *(struct xen_memory_exchange) in* structure store information about the extents that are to be "exchanged". *in.nr_extents* stores the number of extents to be "exchanged"; *in.extent_start* stores the virtual address of the head of an array that contains the memory addresses (GMFNs) of the beginnings of the extents to be "exchanged"; *in.extent_order* stores the order of a single extent.

The fields of the *(struct xen_memory_exchange) out* structure store information about the extents requested from the hypervisor. *out.nr_extents* stores the number of requested extents; *out.extent_order* stores the order of a single requested extent; *out.extent_start* stores the virtual address of the head of an array that consists of GPFNs at which the requested extents are to be mapped in guest VM's memory.

Output:[3] On success, *XENMEM_exchange* returns 0. The array that starts at the address stored in *(struct xen_memory_exchange) out.extent_start* is populated with the memory addresses (GMFNs) of the beginnings of the extents allocated by the hypervisor for the "exchange" operation. On failure, *XENMEM_exchange* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[3]

```
do_memory_op (XENMEM_exchange, (struct xen_memory_exchange) exch)
  call memory_exchange (XENMEM_exchange, (struct xen_memory_exchange) exch)
    . . .
    allocate extent(s) of 2^exch.in.order pages
    store the addresses (GMFNs) of the beginnings of the allocated extents in array mfn
    . . .
    call __copy_to_guest_offset(...)
         populate memory beginning at exch.out.extent_start with the GMFNs in mfn
    return
    . . .
  return
return
```

Description of the vulnerability: The function *__copy_to_guest_offset(to, offset, from, size)*, which is invoked in the handler of the *XENMEM_exchange* hypercall operation, copies data from a virtual address in hypervisor context (*from*) to a virtual address in guest VM context (*to*). For the sake of performance, *__copy_to_guest_offset(to, offset, from, size)* did not perform value validation of the *from* and *to* parameters. As a result, a malicious VM user can invoke *__copy_to_guest_offset(to, offset, from, size)* such that *to* is an address reserved for use by the hypervisor, which leads to overwriting hypervisor's memory. CVE-2012-5513 can be triggered by invoking the *XENMEM_exchange* hypercall operation with an address reserved for use by the hypervisor stored in the *(struct xen_memory_exchange) out.extent_start* parameter.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-5513 was released on 3 December 2012 and is available at [15]. The patch inserts an invocation of the function *guest_handle_okay* in the handler of the *XENMEM_exchange* hypercall operation, which validates the values of the *from* and *to* parameters of *__copy_to_guest_offset*. For instance, a valid virtual address is an address that is not reserved for use by the hypervisor.

Triggering CVE-2012-5513: We triggered CVE-2012-5513 in the following environment:

- guest VM - OS: Debian Squeeze (64 bit), kernel 2.6.32-5-amd64;

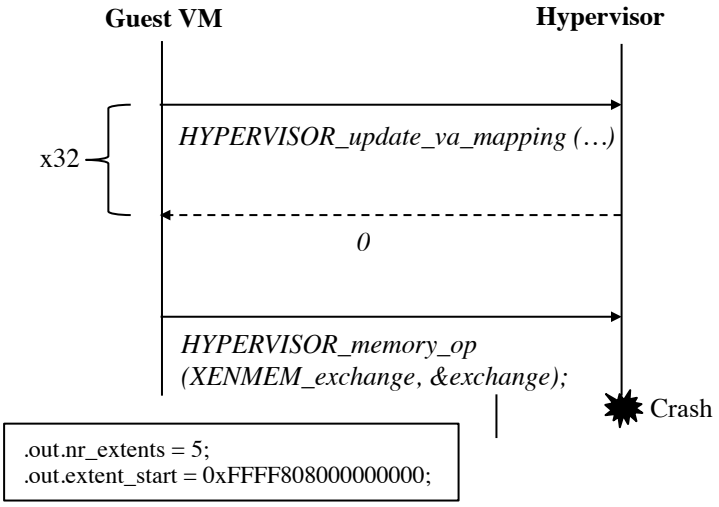

Figure 2.2: An attack triggering CVE-2012-5513

- host VM - OS: Debian Squeeze (64 bit), kernel 2.6.32-5-amd64;
- hypervisor - Xen 4.1.0.

The attack that we executed is depicted in Figure 2.2.

Post-attack state of the hypervisor: When CVE-2012-5513 is triggered, the memory region of the hypervisor beginning at the address stored in *(struct xen_memory_exchange) out.extent_start* is overwritten with the memory addresses (GMFNs) of the beginnings of the extents allocated by the hypervisor for the "exchange" operation. Thus, an attacker cannot control the values with which the hypervisor's memory is overwritten. The amount of data written to the hypervisor's memory is *(struct xen_memory_exchange) out.nr_extents* bytes.

Triggering CVE-2012-5513 may result in a crash of the hypervisor or corrupting its state. Whether the hypervisor crashes depends on which region of the hypervisor's memory is overwritten. An attacker can specify a memory region for overwriting by storing values in the parameters *(struct xen_memory_exchange) out.extent_start* and *(struct xen_memory_exchange) out.nr_extents*. For instance, when we triggered CVE-2012-5513 in our testbed environment, for the values of 0xFFFF808000000000 and 32, and 0xFFFF808000000000 and 16, of *(struct xen_memory_exchange) out.extent_start* and *(struct xen_memory_exchange) out.nr_extents*, respectively, the hypervisor crashed. For the values of 0xFFFF808000000000 and 8 of *(struct xen_memory_exchange) out.extent_start* and *(struct xen_memory_exchange) out.nr_extents*, the hypervisor continued operating with its memory overwritten.

2.2 Hypercall gnttab_op

The *gnttab_op* hypercall is used for managing grant tables. *Grant tables* provide a mechanism for sharing memory between guest VMs (*domains* in Xen terminology) running on top of a Xen hypervisor; that is, it enables the sharing of page frames by granting page frame access permissions to domains or transferring ownerships of pages between domains. Each domain maintains a grant table, which is shared with the hypervisor. A grant table consists of *grant table entries* (i.e., *grants*) indexed by grant references (i.e., *grefs*). In order to access a page frame for which it needs an access permission, a domain first has to *acquire* the grant that grants the access permission from the domain that has issued the grant. When an acquired grant is not needed anymore, it is *released*.

There are version 1 and version 2 grant tables. The format of a grant table entry of a version 1 grant table is *[gref][domid][frame][flags]*, where *gref* is a grant reference, *domid* is the

identification number of domain to which permissions are granted, *frame* is the MFN of the page frame for which permissions are granted, and *flags* are the permissions granted (e.g., read, write, or read and write permissions), which are also referred to as *status* of a grant table entry.

Grant tables of version 2, in addition to grants of the format mentioned above, support transitive grants. *Transitive grants* are used for granting transitive permissions such that a domain issues a grant that refers to a grant issued by another domain.

For the sake of performance, the status of grant table entries of a grant table of version 2 are stored in *status frames*, which are separate from the frames where the rest of the grant table entries are stored.

There are shared and active grants. *Shared grants* are grants issued by a domain. *Active grants* are grants that are in use (i.e., that are acquired) at a given time. A transitive active grant has the fields *trans_domain* and *trans_gref*, where *trans_domain* is the domain that has issued the grant to which the transitive grant refers, and *trans_gref* is the reference of the grant to which the transitive grant refers.

For in-depth information on the grant table mechanism of the Xen hypervisor we refer the reader to [18] and [21].

Vulnerability CVE-2012-4539

> "Xen 4.0 through 4.2, when running 32-bit x86 PV guests on 64-bit hypervisors, allows local guest OS administrators to cause a denial of service (infinite loop and hang or crash) via invalid arguments to GNTTABOP_get_status_frames, aka Grant table hypercall infinite loop DoS vulnerability." [4]

GNTTABOP_get_status_frames is an operation of the *grant_table_op* hypercall, which is used for retrieving MFNs of status frames (i.e., status frame MFNs) of a domain.

Input:[4] *GNTTABOP_get_status_frames* takes as input a structure of type *gnttab_get_status_frames* defined as:

```
struct gnttab_get_status_frames {
  uint32_t nr_frames;
  domid_t dom;
  int16_t status;
  XEN_GUEST_HANDLE(uint64_t) frame_list;
}
```

nr_frames stores the number of requested status frame MFNs; *dom* stores the identification number of the domain whose status frame MFNs are requested; *frame_list* stores the virtual address of the head of an array where status frame MFNs are to be stored upon successful completion of the *GNTTABOP_get_status_frames* operation.

Output:[4] On success, a return code is stored in *(struct gnttab_get_status_frames) status* and the list starting at the address stored in *struct gnttab_get_status_frames) frame_list* is populated with status frame MFNs. On failure, *XENMEM_populate_physmap* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[4]

```
compat_grant_table_op(GNTTABOP_get_status_frames, (struct gnttab_get_status_frames) gf, int count
    = 1)
  rc = 0
  i = 0
  for i < count and rc = 0
    . . .
```

[4]As in Xen of version 4.1.2.

```
    if count = 1
        call gnttab_get_status_frames(gf, ...)
            . . .
            if gf.nr_frames > the number of status frames of domain gf.dom
                gf.status = GNTST_general_error
            else
                . . .
                gf.status = GNTST_okay
        return
        if gf.status = GNTST_okay
            increment i to gf.nr_frames
        . . .
return
```

Description of the vulnerability: In the hypercall handler *compat_grant_table_op*, a *for* cycle loops until the value of the variable *i*, which is initialized to 0, is smaller than the value of the input parameter *count*, which has to be 1. In *compat_grant_table_op*, the value of the variable *i* is incremented to the value of the input parameter *(struct gnttab_get_status_frames) nr_frames* only if *(struct gnttab_get_status_frames) status* stores the value of the constant variable *GNTST_okay*. The value of *(struct gnttab_get_status_frames) status* is set in the function *gnttab_get_status_frames*, which is invoked in *compat_grant_table_op*. *gnttab_get_status_frames* sets the value of *(struct gnttab_get_status_frames) status* to the value of *GNTST_okay* only if the value of the input parameter *(struct gnttab_get_status_frames) nr_frames* is smaller than the number of status frames of the domain whose identification number is stored in the parameter *(struct gnttab_get_status_frames) dom*.

CVE-2012-4539 can be triggered by invoking *GNTTABOP_get_status_frames* such that the value of the input parameter *(struct gnttab_get_status_frames) nr_frames* is greater than the number of status frames of the domain whose identification number is stored in the parameter *(struct gnttab_get_status_frames) dom*. This results in infinite looping of the *for* cycle in *compat_grant_table_op*.

In order to trigger CVE-2012-4539, one has to set the value of *(struct gnttab_get_status_frames) nr_frames* to a value greater than $\lceil \frac{nr\_grants \times sizeof(uint16\_t)}{PAGE\_SIZE} \rceil$, where *nr_grants* is the number of grants issued by the domain whose identification number is stored in *(struct gnttab_get_status_frames) dom*, *PAGE_SIZE* is the size of a single page of the domain, and *uint16_t* is the size of a variable of type unsigned 16-bit integer.

Since the erroneous code is in the handler *compat_grant_table_op*, CVE-2012-4539 can be triggered only from a 64-bit guest VM running on top of a 32-bit host VM.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-4539 was released on 13 November 2012 and is available at [13]. The patch modifies *compat_grant_table_op* such that the value of *i* is set to 1, which is equal to the value of *count*, if the value of *(struct gnttab_get_status_frames) status* is not equal to the value of *GNTST_okay*. This prevents the *for* cycle in *compat_grant_table_op* from looping indefinitely.

Triggering CVE-2012-4539: We triggered CVE-2012-4539 in the following environment:

- guest VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- host VM - OS: Ubuntu Precise (64 bit), kernel 3.8.0-29-generic;
- hypervisor - Xen 4.1.2.

The attack that we executed is depicted in Figure 2.3.

Post-attack state of the hypervisor: When we triggered CVE-2012-4539 in our testbed environment, the guest VM from where we invoked *GNTTABOP_get_status_frames* hanged. When we issued the *xm/xl destroy* command to shutdown the non-responsive guest VM, the

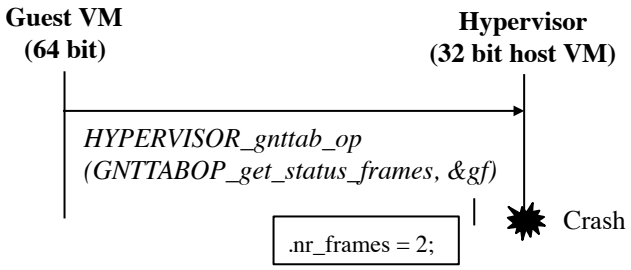

Figure 2.3: An attack triggering CVE-2012-4539

hypervisor crashed. The hypervisor did not crash when we issued the *xm/xl shutdown* command to shutdown, and the *xm/xl reboot* command to reboot, the non-responsive guest VM.

Vulnerability CVE-2012-5510

> "Xen 4.x, when downgrading the grant table version, does not properly remove the status page from the tracking list when freeing the page, which allows local guest OS administrators to cause a denial of service (hypervisor crash) via unspecified vectors." [5]

The *GNTTABOP_set_version* is an operation of the *grant_table_op* hypercall, which is used for downgrading (from version 2 to version 1) or upgrading (from version 1 to version 2) grant tables.

Input:[5] *GNTTABOP_set_version* takes as input a structure of type *gnttab_set_version* defined as:

```
struct gnttab_set_version {
  uint32_t version;
}
```

version stores the version to which the grant table of the domain from where *GNTTABOP_set_version* is invoked is to be set.

Output:[5] On success, *GNTTABOP_set_version* returns 0 and the version of the grant table from where *GNTTABOP_set_version* has been invoked is stored in *(struct gnttab_set_version) version*. On failure, *GNTTABOP_set_version* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[5]

```
do_grant_table_op(GNTTABOP_set_version, ...)
  call gnttab_set_version(...)
    . . .
      if upgrading grant table
        call gnttab_populate_status_frames(...)
            allocate status frames
        return
      if downgrading grant table
        call gnttab_unpopulate_status_frames(...)
            release status frames
        return
    . . .
  return
return
```

[5]As in Xen of version 4.1.2.

Description of the vulnerability: The function *gnttab_unpopulate_status_frames*, which is invoked in the handler of the *GNTTABOP_set_version* hypercall operation, releases allocated status frames when a grant table is downgraded. However, *gnttab_unpopulate_status_frames* does not fully perform the procedure for releasing status frames; that is, it does not remove the nodes that are associated with the status frames being released from the *xenpage_list* linked list. *xenpage_list* is a list of nodes that contain information about frames allocated from the hypervisor's heap memory space for the needs of a given guest VM.

Since *gnttab_unpopulate_status_frames* does not remove from *xenpage_list* the nodes associated with the status frames, subsequent allocation of the same frames leads to adding nodes to *xenpage_list* that are duplicates of the nodes that have not been removed by *gnttab_unpopulate_status_frames*. This is effectively a corruption of *xenpage_list*. The *gnttab_populate_status_frames* function, which is invoked in the handler of *GNTTABOP_set_version* when a grant table is upgraded, may be used for allocating the same frames that have been released when a grant table has been downgraded.

CVE-2012-5510 can be triggered by continuously allocating and releasing status frames, which eventually leads to corruption of *xenpage_list*; that is, CVE-2012-5510 can be triggered by continuously upgrading and downgrading a grant table. When a corruption of *xenpage_list* occurs depends on the amount of free heap memory of the targeted hypervisor as well as the memory allocating mechanism used.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-5510 was released on 3 December 2012 and is available at [14]. The patch modifies the function *gnttab_unpopulate_status_frames* such that it inserts an invocation of the function *put_page*. *put_page* removes from *xenpage_list* the nodes associated with the status frames being released when a grant table is downgraded.

Triggering CVE-2012-5510: We triggered CVE-2012-5510 in the following environment:

- guest VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- host VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- hypervisor - Xen 4.1.2.

The attack that we executed is depicted in Figure 2.4.

Post-attack state of the hypervisor: Depending on the use of *xenpage_list* after it has been corrupted, triggering CVE-2012-5510 may result in crash of the targeted hypervisor or may corrupt its state. The hypervisor crashed when we triggered CVE-2012-5510 in our testbed environment.

Vulnerability CVE-2013-1964

> "Xen 4.0.x and 4.1.x incorrectly releases a grant reference when releasing a non-v1, non-transitive grant, which allows local guest administrators to cause a denial of service (host crash), obtain sensitive information, or possible have other impacts via unspecified vectors." [8]

GNTTABOP_copy is an operation of the *grant_table_op* hypercall, which is used for copying memory pages from a source domain (SD) (i.e., the domain to which the page being copied is allocated) to a destination domain (DD) (i.e., the domain to which the page is copied) with respect to the data read and write permissions set by the SD and/or the DD using grant tables. *GNTTABOP_copy* can be invoked from the SD, the DD, or a domain that is neither the SD or the DD. The domain from where *GNTTABOP_copy* is invoked is called the *local* domain, whereas the other domains involved in copying pages are called *remote* domains.

Input:[6] GNTTABOP_copy takes as input a structure of type *gnttab_copy* defined as:

[6] As in Xen of version 4.1.2.

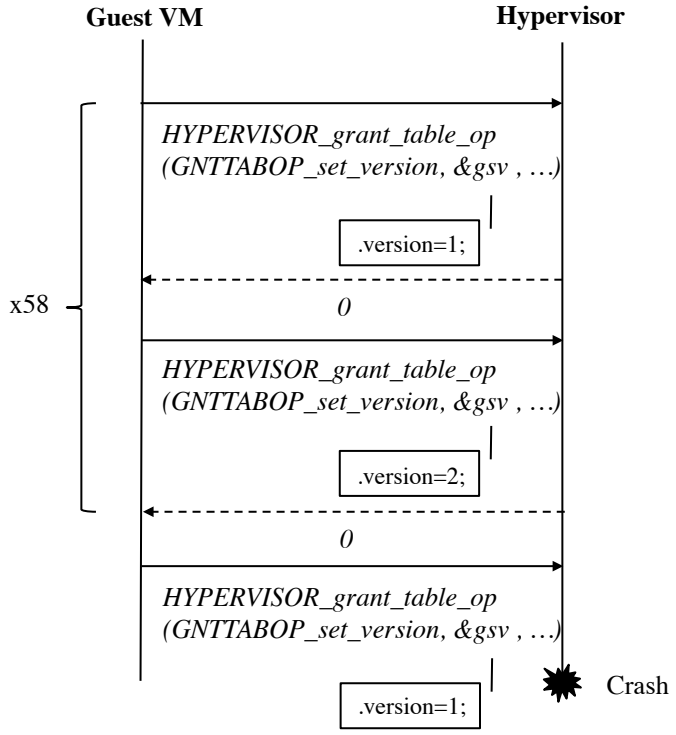

Figure 2.4: An attack triggering CVE-2012-5510

```
struct gnttab_copy {
  struct {
    union {
      grant_ref_t ref;
      xen_pfn_t gmfn;
    } u;
    domid_t domid;
    . . .
  } source, dest;
  uint16_t len;
  uint16_t flags;
  int16_t status;
}
```

source.u.gmfn stores the GMFN of the page that is to be copied if the SD is a local domain; *dest.u.gmfn* stores the GMFN of the page of the DD to which a page of the SD is to be copied if the DD is a local domain; *source.u.ref* stores the grant reference of the grant that grants access to the page that is to be copied if the SD is a remote domain; *dest.u.ref* stores the grant reference of the grant that grants access to the page of the DD to which a page from the SD is to be copied in case the DD is a remote domain; *(source./dest.)u.domid* stores the identification number of the SD/DD; *len* stores the number of bytes to be copied; *flags* stores a value indicating whether the SD and the DD are local or remote domains.

Output:[6] On success, *GNTTABOP_copy* returns 0. On failure, *GNTTABOP_copy* returns an error code (typically a negative integer value). *(struct gnttab_copy)status* stores a value indicating the status of the page copying operation.

Workflow of the vulnerable hypercall handler:[6]

```
i ← the domain invoking GNTTABOP_copy
```

```
d ← the DD

do_grant_table_op(GNTTABOP_copy, struct grant_table_op op, ...)
  call gnttab_copy(op, ...)
    call __gnttab_copy(op, ...)
      . . .
      if the DD is remote
        call __acquire_grant_for_copy
          . . .
          act = active grant table entry (op.dest.ref)
          . . .
          if the grant to be acquired is non−transitive
            . . .
            act.trans_domain = i
            act.trans_gref = 0
          . . .
        return
      . . .
      if the DD is remote
        call __release_grant_for_copy(d, op.dest.ref, ...)
          . . .
          act = active grant table entry (op.dest.ref)
          . . .
          if the grant to be released is of version 2
            if act.trans_domain != d
              call __release_grant_for_copy(act.trans_domid, act.trans_gref, ...)
        return
      . . .
    return
  return
return
```

Description of the vulnerability: In the handler of the hypercall operation *GNTTABOP_copy*, the function *__acquire_grant_for_copy* is used for acquiring grants and *__release_grant_for_copy(d, gref, ...)* for releasing grants, where *d* is the domain that has issued the grant to be released and *gref* is the reference of the grant to be released. In case a grant of version 2 is acquired, the hypervisor creates an active grant and sets the values of its fields *trans_domid* and *trans_gref* to the identification number of the domain from where GNTTABOP_copy has been invoked and 0, respectively. The reason for the latter is to enable scenarios involving, as described in the source code of the handler of *GNTTABOP_copy*, "grant being issued by one domain, sent to another one, and then transitively granted back to the original domain".

The way in which the scenario mentioned above is supported causes non-transitive grants of version 2 to be released as if they were transitive grants (i.e., in a recursive manner). The culprit of this error is that when releasing a grant in the handler of *GNTTABOP_copy*, it is assumed that a transitive grant is a grant whose *trans_dom* field stores a domain identification number that is not equal to the identification number of the domain that has issued the grant being released. However, since the value of the field *trans_domid* of a non-transitive grant is set to the identification number of the domain from where GNTTABOP_copy has been invoked when the grant has been acquired, the previously mentioned condition is also true for non-transitive grants of version 2. As a result, when a non-transitive (active) grant of version 2 is released in the handler of *GNTTABOP_copy*, at least one more grant release takes place, where the grant with a grant reference 0, issued by the domain from where GNTTABOP_copy has been invoked, is released.

An attacker can trigger CVE-2013-1964 by invoking GNTTABOP_copy such that, for example,

a page is copied from a local SD to a remote DD, which has issued a non-transitive grant of version 2.

Vulnerability fix: A patch fixing the vulnerability CVE-2013-1964 was released on 18 April 2013 and is available at [17]. The patch modifies *__acquire_grant_for_copy* such that the value of *trans_domid* is set to the identification number of the domain that issued the grant that is acquired. Further, the value of *trans_gref* is set to the reference of the grant that is acquired. These modifications of *__acquire_grant_for_copy* prevent the recursive release of non-transitive grant of version 2.

Triggering CVE-2013-1964: We triggered CVE-2013-1964 in the following environment:

- guest VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- host VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- hypervisor - Xen 4.1.2.

The attack that we executed is depicted in Figure 2.5.

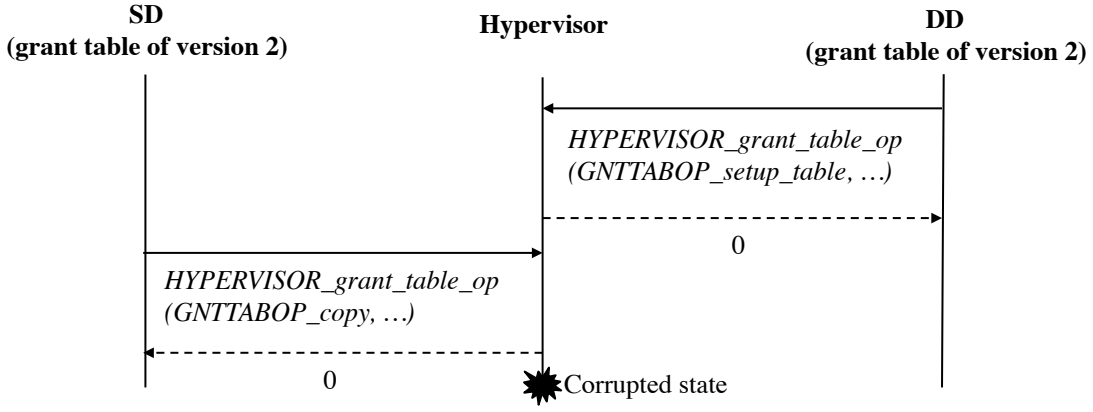

Figure 2.5: An attack triggering CVE-2013-1964

Post-attack state of the hypervisor: Triggering CVE-2013-1964 results in a release of the grant with reference 0 issued by the domain from where GNTTABOP_copy is invoked. Triggering CVE-2013-1964 may disrupt the operation of the hypervisor if the grant released due to the triggering of CVE-2013-1964 is in use (i.e., acquired) at the time of execution of the attack. When we triggered CVE-2013-1964 in our testbed environment, the hypervisor continued operating in a corrupted state.

2.3 Hypercall set_debugreg

Vulnerability CVE-2012-3494

> "The set_debugreg hypercall in include/asm-x86/debugreg.h in Xen 4.0, 4.1, and 4.2, and Citrix XenServer 6.0.2 and earlier, when running on x86-64 systems, allows local OS guest users to cause a denial of service (host crash) by writing to the reserved bits of the DR7 debug control register." [1]

The *set_debugreg* hypercall is used for setting the value of the DR7 register of a CPU allocated to a guest VM. The DR7 register is used for controlling the actions of a CPU when program debugging is performed (e.g., for setting data and/or instruction breakpoints). The addresses at which breakpoints are set in a given debugging session are stored in the registers DR0 - DR3.

The layout of the DR7 register of a 64-bit machine is as follows: ^{bit63} 0 0 0 0 ....0 ^{bit31} [LEN3][R/W3] ... [LEN0][R/W0]^{bit15} 0 0 ^{bit13} GD^{bit11} 0 0 ^{bit9} GE LE ^{bit7} [G3][L3] ... [G0][L0].

The upper 32 bits are reserved and should always be cleared. The *LENx* and *R/Wx* fields are used for specifying the length of the monitored data items when a data breakpoint is set (e.g., 00: one-byte length - also when an instruction breakpoint is set, 01: two-byte length) and the type of program execution break set (e.g., 00 - instruction break, 01 - break on data write, 11 - break on data read and write), respectively. The *GE* (global exact) and/or the *LE* (local exact) bits are set when a data breakpoint is set and instruct the CPU to slow down the execution of the program being debugged so that the exact instruction that triggers the data breakpoint can be reported to the debugging program. The *Gx* and *Lx* bits are used for enabling or disabling breakpoints set at the addresses stored in the registers DR0 - DR3.

Input:[7] *set_debugreg* takes as input a number of a register (an integer value, 7 is used for specifying the DR7 register) and a value that is to be stored in the register (an unsigned long integer value).

Output:[7] On success, *set_debugreg* returns 0. On failure, *set_debugreg* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[7]

```
do_set_debugreg (int reg_nr, unsigned long value)
  call set_debugreg(reg_nr, value)
    if reg_nr = 7
      value &= ~DR_CONTROL_RESERVED_ZERO
      . . .
      store value in DR7
  return
return
```

Description of the vulnerability: In the handler of the *set_debugreg* hypercall, the value of the variable *~DR_CONTROL_RESERVED_ZERO* is applied as a mask with the binary bitwise AND operator to the value of the second parameter of *set_debugreg*. The latter is performed so that the upper 32 bits of the value that is to be stored in the DR7 register are cleared. *DR_CONTROL_RESERVED_ZERO*, which stores the value of *0x0000d800ul*, translates to the binary value of $^{bit63}(0...0)^{bit31}(0000)\ (0000)\ (0000)\ (0000)\ (1101)\ (1000)\ (0000)\ (0000)^{bit0}$. The complement form of the previously mentioned binary number is: $^{bit63}(1...1)^{bit31}(1111)\ (1111)\ (1111)\ (1111)\ (0010)\ (0111)\ (1111)\ (1111)^{bit0}$, which is stored in the variable *~DR_CONTROL_RESERVED_ZERO*. Since they are set to 1, the upper 32 bits of *~DR_CONTROL_RESERVED_ZERO* do not clear the upper 32 bits of the value that is to be stored in the DR7 register when applied as a mask with the binary bitwise AND operator. This results in setting one or multiple bits of the upper 32 bits of the DR7 register to 1, which is not allowed according to hardware specifications.

CVE-2012-3494 can be triggered by invoking *set_debugreg* in a way such that one or multiple bits of the upper 32 bits of the value of the second parameter of *set_debugreg* are set to 1. The bits of the second parameter of *set_debugreg* that are used for setting data or instruction breakpoints (e.g., the bits of the *LENx* fields) should store binary values for setting an instruction breakpoint.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-3494 was released on 5 September 2012 and is available at [10]. The patch assigns the value of *~0xffff27fful* to *DR_CONTROL_RESERVED_ZERO*, and thus, the variable *~DR_CONTROL_RESERVED_ZERO*, which is applied as a mask to the value of the second parameter of *set_debugreg*, has the binary value of $^{bit63}(0...0)^{bit31}(0000)\ (0000)\ (0000)\ (0000)\ (0010)\ (0111)\ (1111)\ (1111)^{bit0}$. Since the upper 32 bits of *~DR_CONTROL_RESERVED_ZERO* are cleared, applying *~DR_CONTROL_RESERVED_ZERO* as a mask to the value of the second

[7] As in Xen of version 4.1.2.

parameter of *set_debugreg* with the binary bitwise AND operator clears the upper 32 bits of the parameter.

Triggering CVE-2012-3494: We triggered CVE-2012-3494 in the following environment:

- guest VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- host VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- hypervisor - Xen 4.1.2.

The attack that we executed is depicted in Figure 2.6.

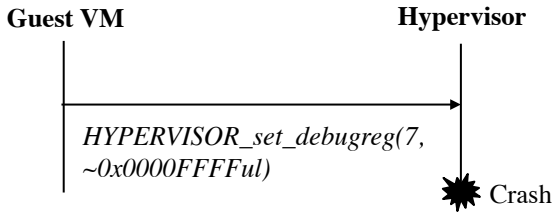

Figure 2.6: An attack triggering CVE-2012-3494

Post-attack state of the hypervisor: Given that in current systems the upper 32 bits of the DR7 register are reserved and should be cleared, triggering CVE-2012-3494 results in crash of the vulnerable hypervisor. However, an outcome different than crash of the hypervisor may be possible if a vulnerable hypervisor is run on future hardware, as stated in [3]: "*if the vulnerable hypervisor is run on future hardware, the impact of the vulnerability might be widened depending on the future assignment of the currently-reserved debug register bits.*"

2.4 Hypercall physdev_op

Vulnerability CVE-2012-3495

> "The physdev_get_free_pirq hypercall in arch/x86/physdev.c in Xen 4.1.x and Citrix XenServer 6.0.2 and earlier uses the return value of the get_free_pirq function as an array index without checking that the return value indicates an error, which allows guest OS users to cause a denial of service (invalid memory write and host crash) and possibly gain privileges via unspecified vectors." [2]

PHYSDEVOP_get_free_pirq is an operation of the *physdev_op* hypercall, which is used for allocating PIRQ (PCI IRQs - Peripheral Component Interconnect Interrupt ReQuests) for the needs of a given guest VM. The Xen hypervisor maintains an array called *pirq_irq* for each guest VM that it hosts. *pirq_irq* is used for marking a given PIRQ as allocated such that the value of the constant variable PIRQ_ALLOCATED (i.e., -1) is stored in the node of *pirq_irq* of index equal to the allocated PIRQ.

Input:[8] *PHYSDEVOP_get_free_pirq* takes as input structure of type *physdev_get_free_pirq* defined as:

```
struct physdev_get_free_pirq {
  int type;
  uint32_t pirq;
}
```

[8] As in Xen of version 4.1.2.

type stores the type of the PIRQ to be allocated (i.e., *MAP_PIRQ_TYPE_GSI* or *MAP_PIRQ_TYPE_MSI*).

Output:[8] On success, *PHYSDEVOP_get_free_pirq* returns 0 and the allocated PIRQ is stored in *(struct physdev_get_free_pirq) pirq*. On failure, -28 is stored in *(struct physdev_get_free_pirq) pirq* and *XENMEM_populate_physmap* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[8]

```
do_physdev_op (PHYSDEVOP_get_free_pirq, (struct physdev_get_free_pirq) gfp)
  . . .
  call gfp.pirq = get_free_pirq(...)
    allocate a PIRQ
  return
  pirq_irq[gfp.pirq] = PIRQ_ALLOCATED
  . . .
return
```

Description of the vulnerability: In the handler of the *PHYSDEVOP_get_free_pirq* hypercall operation, the function *get_free_pirq* is invoked for allocating a PIRQ. The return value of *get_free_pirq* is the allocated PIRQ, if a PIRQ has been succesfully allocated, or an error code (i.e., -28) if a PIRQ could not be allocated. The return value of *get_free_pirq* is used as an index to access an element of the array *pirq_irq* for marking a PIRQ as allocated. However, the return value of *get_free_pirq* is not checked whether it is a PIRQ or an error code. In case *get_free_pirq* returns an error code, the error code is used as an array index and as a result the value of the constant variable *PIRQ_ALLOCATED* (i.e., -1) is written at the memory address &*pirq_irq* - 28, which is a location in hypervisor's memory.

CVE-2012-3495 can be triggered by attempting to allocate a PIRQ when there are no available PIRQs. This can be achieved by invoking the hypercall operation *PHYSDEVOP_get_free_pirq* multiple times until all available PIRQs are allocated and an attempt is made to allocate a PIRQ when there are no available PIRQs. Since PIRQs that can be allocated to a given VM are in the range of 16 to the value of the variable *nr_pirqs_gsi*, a variable in hypervisor context that stores the largest PIRQ that can be allocated to a given VM, invoking *PHYSDEVOP_get_free_pirq* (*nr_pirqs_gsi* - 16) + 2) times is sufficient for triggering CVE-2012-3495.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-3495 was released on 5 September 2012 and is available at [11]. The patch inserts an *if* clause that checks whether the return value of the *get_free_pirq()* function is a PIRQ. If *get_free_pirq()* returns an error code, the error code is not used as an index for accessing an element of the *pirq_irq* array.

Triggering CVE-2012-3495: We triggered CVE-2012-3495 in the following environment:

- guest VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- host VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- hypervisor - Xen 4.1.2.

The attack that we executed is depicted in Figure 2.7.

Post-attack state of the hypervisor: Triggering CVE-2012-3495 results in overwriting the hypervisor's memory at the memory address &pirq_irq - 28 with the value of the variable PIRQ_ALLOCATED (i.e., -1). An attacker cannot control the value written in the hypervisor's memory. Depending on the memory layout of the hypervisor, the hypervisor may crash or continue operating in a corrupted state. When we triggered CVE-2012-3495 in our testbed environment, the hypervisor continued operating in a corrupted state.

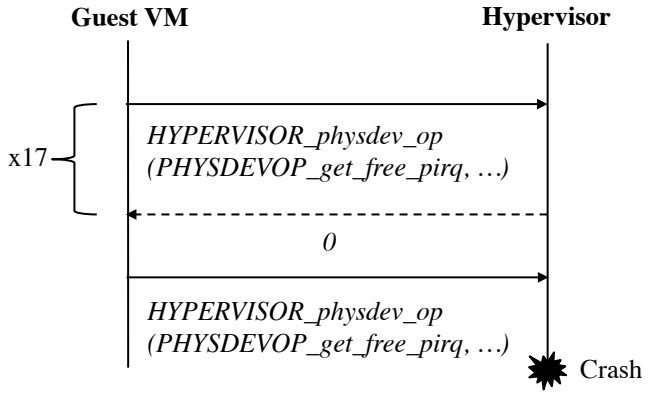

Figure 2.7: An attack triggering CVE-2012-3495

2.5 Hypercall mmuext_op

Vulnerability CVE-2012-5525

> "The *get_page_from_gfn* hypercall function in Xen 4.2 allows local PV guest OS administrators to cause a denial of service (crash) via a crafted GFN that triggers a buffer over-read." [7]

The *get_page_from_gfn* function provides information about a given memory page. It is invoked in the handlers of multiple hypercalls of the Xen hypervisor, one of which is the handler of the *MMUEXT_CLEAR_PAGE* operation of the *mmuext_op* hypercall. *MMUEXT_CLEAR_PAGE* is an operation of the *mmuext_op* hypercall, which is used for clearing memory pages/frames.

Input:[9] *MMUEXT_CLEAR_PAGE* takes as input structure of type *mmuext_op* defined as:

```
struct mmuext_op {
  unsigned int cmd;
  union {
  xen_pfn_t mfn;
  . . .
  } arg1;
  . . .
}
```

cmd stores a number identifying an operation of the *mmuext_op* hypercall (e.g., *MMUEXT_CLEAR_PAGE*); *arg1.mfn* stores the MFN of the page that is to be cleared.

Output:[9] On success, *MMUEXT_CLEAR_PAGE* returns 0. On failure, *MMUEXT_CLEAR_PAGE* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[9]

```
do_mmuext_op ((struct mmuext_op) op, ...)
  struct page_info page;
  call page = get_page_from_gfn(op.arg1.mfn, ...)
  . . .
return
```

Description of the vulnerability: *get_page_from_gfn* reads information about a page allocated to a guest VM from the frame table of the VM using the MFN of the page as offset. A

[9] As in Xen of version 4.2.0.

frame table is a memory area shared between the hypervisor and a guest VM where information about each page allocated to the guest VM is stored in the format of a structure of type *page_info*.

The MFN used by *get_page_from_gfn* for reading page information is provided to *get_page_from_gfn* as an input parameter. The value of the MFN provided as input parameter to *get_page_from_gfn* is not checked for validity. Since *get_page_from_gfn* uses a MFN as an offset for reading from the frame table of a given guest VM, an invalid MFN is a MFN that causes a buffer over-read (i.e., that is larger than the largest MFN at which a page of the guest VM is allocated). An attacker can provide an invalid MFN as an input parameter to *get_page_from_gfn*, in which case *get_page_from_gfn* returns invalid page information.

In the handler of the *MMUEXT_CLEAR_PAGE* hypercall operation, the MFN stored in the input parameter *(struct mmuext_op) arg1.mfn* is provided to *get_page_from_gfn* for reading page information. CVE-2012-5525 can be triggered by invoking *MMUEXT_CLEAR_PAGE* such that an invalid MFN is stored in *(struct mmuext_op) arg1.mfn*.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-5525 was released on 3 December 2012 and is available at [16]. The patch inserts an invocation of the function *mfn_valid* in *get_page_from_gfn*, which verifies the validity of the MFN provided as input to *get_page_from_gfn*. The patch modifies *get_page_from_gfn* such that if the MFN used for reading page information is not valid, *get_page_from_gfn* returns NULL instead of invalid page information.

Triggering CVE-2012-5525: We triggered CVE-2012-5525 in the following environment:

- guest VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- host VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- hypervisor - Xen 4.2.0.

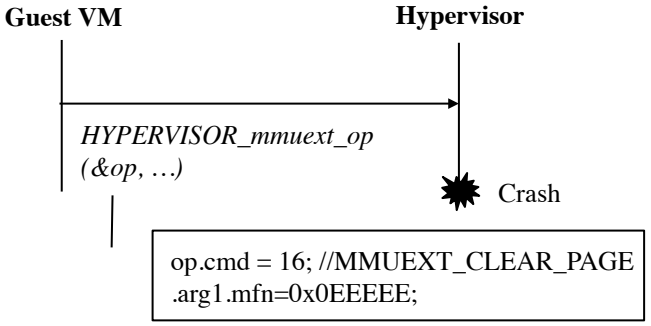

Figure 2.8: An attack triggering CVE-2012-5525

The attack that we executed is depicted in Figure 2.8.

Post-attack state of the hypervisor: Triggering CVE-2012-5525 may result in a crash of the targeted hypervisor or may corrupt its state. Whether the hypervisor crashes depends on the use of the invalid page information returned from *get_page_from_gfn* when CVE-2012-5525 is triggered. The hypervisor crashed when we triggered CVE-2012-5525 in our testbed environment.

References

[1] CVE-2012-3494. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3494.

[2] CVE-2012-3495. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3495.

[3] CVE-2012-3496. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-3496.

[4] CVE-2012-4539. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-4539.

[5] CVE-2012-5510. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5510.

[6] CVE-2012-5513. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5513.

[7] CVE-2012-5525. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-5525.

[8] CVE-2013-1964. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-1964.

[9] Xen Interface Manual. http://www-archive.xenproject.org/files/xen_interface.pdf.

[10] Xen Security Advisory 12 (CVE-2012-3494). http://lists.xen.org/archives/html/xen-announce/2012-09/msg00000.html.

[11] Xen Security Advisory 13 (CVE-2012-3495). http://lists.xen.org/archives/html/xen-announce/2012-09/msg00001.html.

[12] Xen Security Advisory 14 (CVE-2012-3496). http://lists.xen.org/archives/html/xen-announce/2012-09/msg00002.html.

[13] Xen Security Advisory 24 (CVE-2012-4539). http://lists.xen.org/archives/html/xen-announce/2012-11/msg00002.html.

[14] Xen Security Advisory 26 (CVE-2012-5510). http://lists.xen.org/archives/html/xen-announce/2012-12/msg00001.html.

[15] Xen Security Advisory 29 (CVE-2012-5513). http://lists.xen.org/archives/html/xen-announce/2012-12/msg00004.html.

[16] Xen Security Advisory 32 (CVE-2012-5525). http://lists.xen.org/archives/html/xen-announce/2012-12/msg00002.html.

[17] Xen Security Advisory 50 (CVE-2013-1964). http://lists.xen.org/archives/html/xen-announce/2013-04/msg00006.html.

[18] Xen Wiki. http://wiki.xenproject.org/wiki/Main_Page.

[19] The 2012 ACM Computing Classification System. `http://www.acm.org/about/class/2012`, 2012.

[20] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *SIGOPS Operating Systems Review*, 37(5):164–177, October 2003.

[21] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2007.

[22] CVE Details. http://www.cvedetails.com/.

[23] Frank Gens, Robert Mahowald, L. Richard Villars, David Bradshaw, and Chris Morris. Cloud Computing 2010: An IDC Update, 2010.

[24] Avi Kivity. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[25] Joanna Rutkowska and Rafał Wojtczuk. Xen 0wning Trilogy: Part Two (presentation slides). http://invisiblethingslab.com/resources/bh08/part2.pdf.