Sabotaging the System Boundary: A Study of the Inter-boundary Vulnerability

Pengfei Wang[a,*], Xu Zhou[a] and Kai Lu[a]

[a]*College of Computer, National University of Defense Technology, Changsha 410073, China.*

ARTICLE INFO

Keywords:
Inter-boundary vulnerability
System boundary
Shared memory bug
Real-world case investigation
Double-fetch vulnerability

ABSTRACT

The hierarchy theory is the foundation of the modern computer system design. However, the interaction part between different system layers is usually the weak point of the system, which tends to have security flaws. When communicating across the system boundary, failure to enforce the required synchronization in the shared memory can cause data inconsistency of the communication partners. Especially when there is a privilege gap between different boundary sides, such data inconsistency can lead to security vulnerability and sabotage the trust boundary. In this paper, we propose the concept of inter-boundary vulnerability and give the first in-depth study of them. We investigate three typical boundaries in the system that inter-boundary vulnerabilities are prone to occur, including the kernel-user boundary, the hardware-OS boundary, and the VMM-guest OS boundary. Then, based on the investigation of 115 real-world vulnerability cases, we extract four vulnerability types and provide analysis for each type to illustrate the principle. Finally, we discuss the state-of-the-art techniques that are relevant to the detection, prevention, and exploitation of such vulnerabilities, aiming to light the future research on this topic.

1. Introduction

The hierarchical structure is the foundation of the computer system, which facilitates system design and implementation. However, owing to the complex functionality, such as data exchange, privilege isolation, and error disposal, the interaction part between different system layers is usually the weakness of the system. Besides, modern optimization schemes, such as concurrent processing and asynchrionization communication, also enlarge the security risks. Thus, communication between different system layers tends to have security flaws.

Shared memory is a fundamental and widespread communication scheme for the inter-domain communication of modern computer systems. The main reason for its popularity is the performance advantage compared to the other message-based communication mechanisms. However, the shared memory scheme is also vulnerable when used for cross-layer communication. Communication-based on shared memory usually requires additional synchronization, such as locks and mutexes. Otherwise, concurrent access to shared data can cause memory corruption errors. These synchronization methods require all communication partners to participate, otherwise, the synchronization cannot be enforced. This usually is not a problem when all communication participants operate on the same privilege level, such as communication between normal processes or between threads. However, when one side of the communication is less privileged, the shared memory interface becomes a trust boundary, and the situation becomes complicated [1]. Since high-level synchronization methods are not enforced in shared memory interfaces, they can simply be ignored, causing data inconsistency. Such data inconsistency can sabotage the trust boundary and cause security vulnerability, which can lead to severe consequences, such as memory corruption errors and sensitive information disclosure. We call it the "inter-boundary vulnerability".

The inter-boundary vulnerability exists in the communication between different system layers (or domains). Different from the concurrency bugs, the inter-boundary vulnerability usually occurs where there is a privilege gap, known as the trust boundary. Consequently, the misuse of synchronization primitives by one communication partner (usually the privileged one) gives a chance to the less privileged (malicious) partner to cause harmful results to the privileged one. Although there is a large amount of research on the safe use of shared resources, such as the race conditions [2, 3, 4, 5, 6, 7, 8] and concurrency bugs [9, 10, 11, 12, 13, 14, 15], they usually focus on the insecure behavior caused by the misuse of synchronization primitives. They do not take the existence of a malicious communication partner into account, thus, hardly applicable to the detection of security vulnerabilities in the thrust boundary. Therefore, inter-boundary vulnerability is a new research point that supplements these researches.

Previous research has raised the awareness of the double-fetch vulnerability [16, 17, 18, 19, 20]. The double-fetch vulnerability is a subclass of the inter-boundary vulnerability, which is caused by the violation of the read-after-read data dependency between the kernel address space and user address space. However, as we have mentioned above, such vulnerability is theoretically not limited to the kernel-user boundary nor the read-after-read data dependency. Thus, in this paper, we propose the concept of inter-boundary vulnerability and give the first in-depth study of it. We broaden the scope of this topic by studying more system boundary types and analyzing more data dependency types, aiming to light

*Corresponding author
pfwang@nudt.edu.cn (P. Wang)
wpengfei.github.io (P. Wang)
ORCID(s): 0000-0003-3408-4153 (P. Wang)

the future research on this topic. In summary, we make the following contributions.

- We extract three system boundaries that inter-boundary vulnerabilities are prone to occur, including the kernel-user boundary, the hardware-OS boundary, and the VMM-guest OS boundary.
- We investigate 115 real-world inter-boundary vulnerability cases and categorize four inter-boundary vulnerability types based on the analysis of these cases. We have made the collected vulnerabilities available online for the security community for further research.
- We review the state-of-the-art techniques that are relevant to the detection, exploitation, and prevention of the inter-boundary vulnerability, and point out challenges for the future work.

The rest of the paper is organized as follows: Section 2 reviews the background knowledge of this topic and introduces the vulnerability-prone boundaries in the system. Section 3 classifies the inter-boundary vulnerabilities and analyzes related works on detection, exploitation, and prevention. Section 4 gives an in-depth analysis of this topic. Section 5 discusses the perspectives and challenges for the future work, followed by conclusions.

2. Background

2.1. The Shared Memory Scheme

Shared memory is a fundamental and widespread communication scheme for the inter-domain communication of modern computer systems. The main reason for its popularity is the performance advantage compared to the other message-based communication mechanisms, such as pipes or message queues, which are implemented on top of system calls [1].

When the data is transferred between two processes, a message-oriented approach requires at least two additional copies into the kernel as the pipe and message queue resides in the kernel. The sender triggers a copy from the user space to pass data to the kernel, and the receiver uses another copy to get the data from the kernel back into the user space [21]. However, for the shared memory scheme, the procedure is straightforward. It shares the physical memory pages by mapping them into the virtual address space of multiple execution contexts (user space process). The shared memory scheme only has a one-time setup cost. After creating the page-mapping, data transfers between two contexts do not require any involvement of the kerne. Instead, simple memory reads and writes are used, which avoids the expensive copy operations [1]. Thus, it has a performance advantage.

Based on the shared memory, communication usually requires proper synchronization between the communication partners, such as the mutexes, locks, and semaphores. However, an obvious limitation of these synchronization methods are that they require all communication partners to participate, otherwise, the synchronization cannot proceed [1].

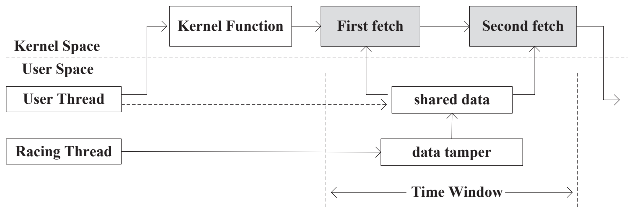


Figure 1: Illustration of how the double-fetch vulnerability in the kernel-user boundary occurs.

This usually is not a problem when all communication participants operate on the same privilege level, such as communication between normal processes or between threads. However, when one side of the communication has less privilege, the shared memory interface becomes a trust boundary, and the situation becomes complicated. From a security perspective, the higher privilege side of the communication needs to protect itself against the malicious behavior of its counterpart. Thus, the privileged code operating on the shared memory needs auditing for security vulnerabilities. Such situations include the communication between the kernel space and user space, the communication between the hardware and the operating system (OS), and the communication between the VMM and the guest OS. Since high-level synchronization methods are not enforced in shared memory interfaces, they can simply be ignored, causing potential vulnerabilities.

2.2. Boundaries in the System

Modern computer systems have a hierarchical structure, and the complex function is divided and implemented in abstracted layers. Each layer (also known as a domain) conducts a relatively centralized and straightforward job to facilitate the system design and implementation. The isolation between the two domains also improves the security scheme, forming a virtual boundary. From a security point of view, a boundary is a privilege gap between two domains. The higher privilege side needs to protect itself against potential malicious behaviors of its counterpart, such as the boundary between the kernel and user address space. In this paper, we choose the kernel-user boundary, the hardware-OS boundary, and the VMM-OS boundary as representatives to give a detailed analysis of the inter-boundary vulnerability.

2.2.1. The Kernel-user Boundary

In the modern computer system, for security purposes, the memory address space is divided into the kernel space and user space. The kernel space is where the kernel code executes, while the user space is where regular user processes run [17]. Although the kernel space and the user space are both virtually and physically isolated, owing to the kernel privilege, some memory data that resides in the user space is accessible by both the kernel and the user process [16].

A typical vulnerability type that crosses the kernel-user boundary is the double-fetch vulnerability [16, 18, 20, 19]. In 2008, Fermin J. Serna [22] first introduced the term“double

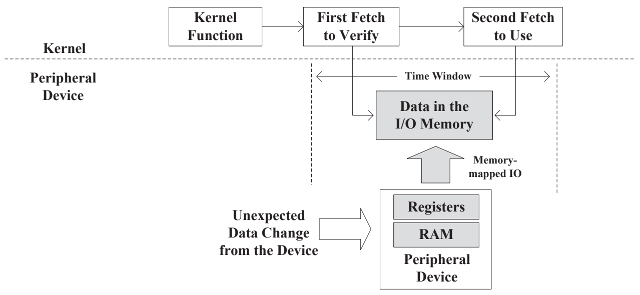


Figure 2: Illustration of how the double-fetch vulnerability in the hardware-OS boundary occurs [25].

fetch" to describe a special kernel vulnerability type that was caused by a race condition between the kernel and user space. A double fetch is a situation that the kernel (e.g., via a syscall) reads the same value that resides in the user space twice, the first time to verify the value and the second time to use the value [23]. As Figure 1 shows, such a situation turns into a double-fetch vulnerability when a concurrently running user thread tampers the value between the two kernel reads under a race condition, then, when the kernel reads the value a second time to use, it gets a different one [17]. The data inconsistency introduced by the double-fetch vulnerability could lead to problematic consequences such as privilege escalation, information leakage, kernel crash, etc [17]. The famous KHOBE attack [24] is a typical application of the double-fetch vulnerability to bypass the security software on the system [17].

The double-fetch vulnerability is different from the typical race condition because the kernel-user boundary separates the race condition in a double-fetch vulnerability. The kernel space only contains two reads, and the write resides in the user space. Moreover, the write from the user space is usually potential, which does not necessarily exist in regular times, but can be crafted by a malicious user to create a race condition when triggering the vulnerability [17].

2.2.2. The Hardware-OS Boundary

In a computer system, a boundary exists between the hardware device and the software. When attaching to the system, the peripheral devices are controlled by the OS. Specifically, the OS controls the hardware device by writing to and reading from its registers [25]. Most of the time, a device has several registers, and they are accessed at consecutive addresses, either in the memory address space or in the I/O address space [25]. The I/O memory is simply a region of RAM-like locations that the device makes available to the processor over the bus. For example, the memory-mapped I/O scheme uses the same address space to address both the physical memory and the I/O devices, and the CPU uses the same instructions used to access the physical memory to access the device registers and device memory [25]. Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the data bus to the desired device's hardware register [25]. Both memory-mapped registers and memory-mapped device

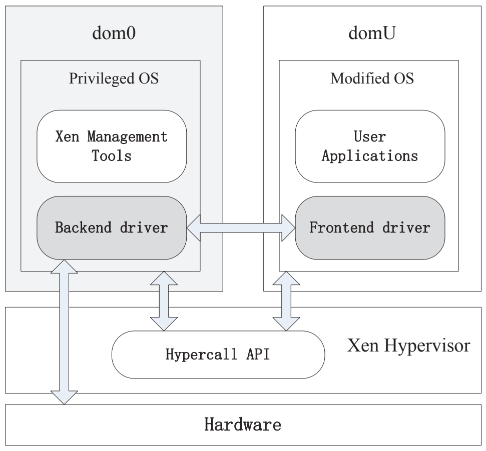


Figure 3: Architecture of the virtualized system.

memory are called the I/O memory because the difference between registers and memory from peripheral devices is transparent to software [25]. Thus, accessing the I/O memory is in the same fashion as accessing the regular memory, and memory-corruption errors can also occur in the I/O memory.

A recent study [25] reveals a new double-fetch vulnerability type that crosses the hardware-OS boundary. As Fig. 2 shows, this vulnerability occurs when the OS controls a peripheral device by accessing the I/O memory. The kernel reads the "same" I/O memory data twice, assuming the data is unchanged. However, since the driver is unable to validate the attached device fully, the compromised hardware could tamper the peripheral data mapped in the I/O memory between the two reads, causing data inconsistency for the OS. Such data inconsistency may lead to severe problems for the system functioning or even a security vulnerability [25].

This double-fetch vulnerability is different from the traditional one because it crosses different boundaries. The involvement of peripheral devices makes the new case a complicated problem. For example, in a traditional double-fetch vulnerability, the kernel fetches data from user space, and the user thread tampers the shared data under race conditions. But in this peripheral type, the tampered data resides in the I/O memory, and the data is modified by the peripheral device.

2.2.3. The VMM-guest OS Boundary

In the virtualized system, a new software layer is introduced, called the virtual machine monitor (VMM) or the hypervisor. Each virtual machine (VM) consists of the virtual memory, the virtual CPUs, and the virtualized devices. The VMM is responsible for managing the accesses from each running virtual machine to the hardware, giving a guest OS the illusion to be running on real physical hardware.

In a virtualized system, the VMM runs in the ring 0 mode, which has access to privileged instructions, the entire memory space, and the I/O. For security purposes, the virtual hardware must isolate from each VM, and the privileged operations of the guest OS must be restricted. There are two ways to realize such restrictions. VMWare uses binary translation to dynamically replaces privileged operations with emulated versions that operate on the virtual hardware [26]. Xen uses para-virtualization to modify the guest OS to replace all privileged operations with calls to the VMM APIs [27]. As a result, the guest OS kernel is moved to a less privileged ring, making the VMM the only code running in ring 0.

As Fig. 3 shows, the Xen hypervisor operates directly on top of the hardware and hosts many virtual machines called domains. Among them, `dom0` is a privileged management domain, running a typical Linux system with all the management tools required for the hypervisor and its guests; `domU` is the unprivileged para-virtualized guest, running a modified guest OS. The guest OS does not interact with the real hardware. Instead, it communicates directly with the hypervisor using the hyper-call API (an interface similar to the regular system call interface). As a result, all privileged functionality is restricted to `dom0`, and the actions performed by the `domU` kernel can only affect its own VM.

The para-virtualized guest uses a split driver model to access the virtual hardware devices, which consists of the frontend and backend components. The frontend driver in `domU` works as a regular hardware device driver in the guest OS. When the virtual device is accessed, the frontend driver sends a request to the backend driver in `dom0`, which processes the request and manipulates the real hardware device [1]. The core mechanism used for inter-domain communication between the backend and frontend components is the shared memory,

Even though the privileges of backend components is restricted to reduce the impact of a vulnerability, for the performance reason, many backend components are directly implemented in the kernel of the management system, making full isolation impossible. Thus, attacks on the backend components of para-virtualized drivers can directly lead to a full `dom0` compromise. Such compromise is as critical as a compromise of the Xen hypervisor itself because `dom0` is a privileged management domain that has access to the complete state of all other guests and communicates directly with the hardware. Thus, the inter-domain communication via shared memory between the frontend and backend drivers is a potential attack surface.

3. Introduction to the Inter-boundary Vulnerability

The inter-boundary vulnerability occurs when different system layers (or domains) communicating via the shared memory scheme. Communication partners access the shared data without proper use of the required synchronization can violate the data consistency and cause memory corruption errors. The privilege gap between the communication partners can turn such memory errors to security vulnerabilities that breach the trust boundary. As Fig. 4 shows, an inter-boundary vulnerability consists of the following five elements:

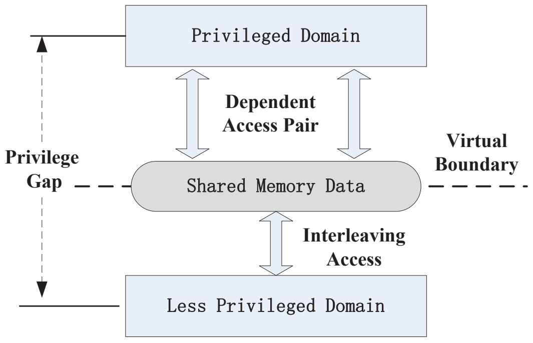


Figure 4: Illustration of the inter-boundary vulnerability.

(1) Isolated domains. Two functional isolated or restricted domains. Isolated domains work in an asynchronized way but with proper communications. For example, the kernel space and user space are isolated by the restriction of the address space. However, they can communicate with each other via various schemes, such as the system calls.

(2) Shared memory data. The isolated domains conduct communication via shared memory data, such as passing arguments and synchronization. The shared data can be global variables, heap objects, or data blocks that are being processed concurrently. Since both of the communication partners can access the shared data, the synchronization is of vital importance. However, it is usually hard to enforce both sides of the communication.

(3) The privilege gap. Since the functionality of each domain is centralized and secured, the isolation between two domains forms a virtual boundary. From a security point of view, a boundary usually has a privilege gap, which protects the privileged domain (such as the kernel space) from the malicious behaviors of the less privileged counterpart (such as the user space).

(4) The dependent access pair. Shared data accessible to both of the two domains. When a domain launches multiple accesses to the shared data, the accesses can be dependent. For instance, two reads can be dependent as the first is used to check the data, while the second is to use the data. Such dependency also includes read-after-write, write-after-read, and double-write. We give a detailed analysis in the next subsection.

(5) The interleaving access. When a domain has a dependent access pair to the shared data, the other domain can launch an interleaving access to the shared data between the access pair. For example, a write from one domain can interleave between two dependent reads from the other domain to tamper the data, causing a buggy result.

3.1. Classification

An inter-boundary vulnerability involves three memory accesses to the same shared variable – two local dependent accesses from the privileged domain and one remote interleaving access from the less privileged domain. When the remote access occurs between the two local accesses, there

Table 1
The theoretical inter-boundary vulnerability types.

Types	Description
R_1 W_i R_2	Two local reads interleaved by a remote write, making the two reads have different views of the same memory location.
W_1 W_i R_2	Local read after write interleaved by a remote write. The local read does not get the local result it expects.
W_1 R_i W_2	Two local writes interleaved by a remote read, leaking the intermediate result between the writes to the remote read.
R_1 W_i W_2	Local write-after-read interleaved by a remote write, which violates the dependency of the local write-after-read.

* The subscripts *1* and *2* indicate the two local accesses in the privileged domain while *i* indicates the interleaving access from the less privileged domain. The *R* and *W* indicate the read and write respectively.

```
 84 static long vbg_misc_device_ioctl(struct file *filp, unsigned int req,
 85                                    unsigned long arg) {
...
 94   if (copy_from_user(&hdr, (void *)arg, sizeof(hdr)))
 95     return -EFAULT;
...
100   if (hdr.size_in < sizeof(hdr) ||
101     (hdr.size_out && hdr.size_out < sizeof(hdr)))
102       return -EINVAL;
103
104    size = max(hdr.size_in, hdr.size_out);
...
117   if (is_vmmdev_req)
118     buf = vbg_req_alloc(size, VBG_IOCTL_HDR_TYPE_DEFAULT);
119   else
120     buf = kmalloc(size, GFP_KERNEL);
121   if (!buf)
122     return -ENOMEM;
123
124   if (copy_from_user(buf, (void *)arg, hdr.size_in)) {
125     ret = -EFAULT;
126     goto out;
127   }
128   if (hdr.size_in < size)
129     memset(buf + hdr.size_in, 0, size -  hdr.size_in);
130
131   ret = vbg_core_ioctl(session, req, buf);
... }
```

Figure 5: A double-read type inter-boundary vulnerability in the kernel-user boundary.

should theoretically form eight combinations. However, only four of them can cause buggy results, shown in Table 1. Accordingly, we categorize the inter-boundary vulnerabilities into four types: the double-read (shorted as R-R) type, the read-after-write (shorted as W-R) type, the double-write (shorted as W-W) type, and the write-after-read (shorted as R-W) type. In the following subsections, we use real-world cases to illustrate how each type of vulnerability occurs.

3.1.1. The Double-read Type

The double-read type, also known as the double-fetch vulnerability, is the most prevalent inter-boundary vulnerability type, which has manifested in all the three boundary types.

Figure 5 shows a double-read type inter-boundary vulnerability in the kernel-user boundary (CVE-2018-12633), which occurs in `/drivers/virt/vboxguest/vboxguest_linux.c` of Linux-4.16.8. In function `vbg_misc_device_ioctl()`, the driver reads the same user data twice by `copy_from_user()`. The first read copies the message header (stored in `hdr`), which is used to verify the size variables (line 100-102), such as `hdr.size_in` and `hdr.size_out`. Then, a kernel buffer is allocated based on variable `size` (line 117-122). In the second read (line 124), the whole message is copied into the buffer, and the header part is copied again. Though the header part is double-fetched, the critical variables such as `hdr.size_in` and `hdr.size_out` are only verified after the first read but not after the second read. Thus, the user data could be tampered between the two reads by a concurrently running malicious user thread under a race condition. Since the kernel buffer is allocated based on the variables from the first read, whereas the buffer is used with variables from the second read in `vbg_core_ioctl()` (line 131), the malicious data tamper from the user side can lead to a kernel buffer over-access.

```
118 void snd_msndmidi_input_read(void *mpuv){
...
126   while (readw(mpu->dev->MIDQ+JQS_wTail) != readw(mpu->dev->MIDQ+JQS_wHead)) {
127       u16 wTmp, val;
128       val = readw(pwMIDQData + 2 * readw(mpu->dev->MIDQ + JQS_wHead));
129
130       if (test_bit(MSNDMIDI_MODE_BIT_INPUT_TRIGGER, &mpu->mode))
132         snd_rawmidi_receive(mpu->substream_input,(unsigned char *)&val, 1);
...
135       wTmp = readw(mpu->dev->MIDQ + JQS_wHead) + 1;
136       if (wTmp > readw(mpu->dev->MIDQ + JQS_wSize))
137         writew(0,  mpu->dev->MIDQ + JQS_wHead);
138       else
139         writew(wTmp,  mpu->dev->MIDQ + JQS_wHead);
140   }
142 }
```

Figure 6: A double-read type inter-boundary vulnerability in the hardware-OS boundary.

Figure 6 shows a double-read type inter-boundary vulnerability in the hardware-OS boundary (CVE-2017-9985), which occurs in file `/sound/isa/msnd/msnd_midi.c` of Linux-4.10.1. The kernel uses wrapper function `readw()` to read hardware device data from the I/O memory to the kernel. Function `snd_msndmidi_input_read()` uses a header pointer `mpu->dev->MIDQ + JQS_wHead` to process a message queue, and pointer value is fetched twice at line 126 and line 128, respectively. The first time is to prevent queue over-access by comparison with the tail pointer, whereas the second time is to get the message data from the queue and send it by `snd_rawmidi_receive()`. This double-fetch situation of the header pointer is vulnerable because the pointer resides in the peripheral device register, and the value of the pointer is likely to be changed between the two reads of the pointer. Once the pointer value is tampered by unexpected data flips or intentionally compromised hardware, an array (or queue) over-access could occur.

Figure 7 shows a simplified version of a double-read type inter-boundary vulnerability in the VMM-guest OS boundary, which affects xen_disk, a block backend implementation in Xen. Xen defines helper function `blkif_get_x86_64_req()`

```
1 void blkif_get_x86_64_req(blkif_request_t *dst,
                            blkif_x86_64_request_t *src){
2     int i, n = BLKIF_MAX_SEGMENTS_PER_REQUEST;
3
4     dst->operation = src->operation;
5     dst->nr_segments = src->nr_segments;
6     // ...
7     if (src->operation == BLKIF_OP_DISCARD) {
8         //..
9     }
10    if (n > src->nr_segments)
11    n = src->nr_segments;
12    for (i = 0; i < n; i++)
13    dst->seg[i] = src->seg[i];
14 }
```

Figure 7: A double-read type inter-boundary vulnerability in the VMM-guest OS boundary.

```
1 NTSTATUS NTAPI NtGetSystemVersion(
2     PUNICODE_STRING UnicodeString,
3     PWCHAR Buffer,
4     DWORD BufferLength
5 ) {
6     UnicodeString->Length = 0;
7     UnicodeString->MaximumLength = BufferLength;
8     UnicodeString->Buffer = Buffer;
9
10    RtlAppendUnicodeToString(UnicodeString,
11    L"Microsoft␣Windows␣[Version␣10.0.16299]");
12
13    return STATUS_SUCCESS;
14 }
```

Figure 8: A read-after-write type inter-boundary vulnerability in the kernel-user boundary.

to parse and copy frontend requests from the shared memory to a private buffer. Variable `src` points into the shared memory between the frontend and backend driver, and two consecutive accesses (lines 10 and 11) to the `nr_segments` field of `src` form a typical scene of a double-fetch vulnerability. A malicious user can bypass the check at line 10 and trigger a heap overflow in the backend by tampering the value of `src->nr_segments` between line 10 and 11 in the shared memory. The exploitation works by setting `src->nr_segments` to a smaller value than `n` to pass the condition check, but then change it to a greater value before line 11. Consequently, it could cause an overflow in the loop afterward [1].

3.1.2. The Read-after-write Type

The read-after-write type is very similar to the double-read type. A privileged domain writes to a memory address in a less privileged domain and subsequently reads from it with the assumption that the written value has remained unchanged, however, the value could have been changed.

Figure 8 shows a read-after-write type inter-boundary vulnerability that crosses the kernel-user boundary [28]. The syscall handler initializes object `UnicodeString` at lines 6-8. Then, it uses `RtlAppendUnicodeString` to fill the object with a textual representation of the system version. However, this procedure is problematic because the latter routine assumes the object it receives is non-volatile, but in fact, the object contains a user-mode pointer `Buffer` whose data may change

```
1 typedef struct _USERNAME {
2     UNICODE_STRING String;
3     WCHAR Buf[128];
4 } USERNAME, *PUSERNAME;
5
6 NTSTATUS NTAPI NtGetAdminUsername(PUSERNAME OutputUser) {
7   USERNAME LocalUser;
8
9   RtlZeroMemory(&LocalUser, sizeof(USERNAME));
10
11  StringCchCopy(LocalUser.Buf, 128, "Administrator");
12  RtlInitUnicodeString(&LocalUser.String, LocalUser.Buf);
13
14  RtlCopyMemory(OutputUser, &LocalUser, sizeof(USERNAME));
15  OutputUser->String.Buf = OutputUser->Buf;
16
17  return STATUS_SUCCESS;
18 }
```

Figure 9: A double-write type inter-boundary vulnerability in the kernel-user boundary.

asynchronously during the syscall execution. This situation can be exploited by a malicious concurrent thread under a race condition, changing the value of `UnicodeString->Buffer` to the kernel address space within the time window between the initialization of the pointer and its usage.

In the read-after-write type, the vulnerable code trusts the contents of user-mode memory not because it has already read it once (like the double-read type), but because it has explicitly initialized it to a specific value [28]. However, this vulnerability type is observed to be Windows-specific because it is strongly tied to the direct user-mode pointer manipulation in Windows [1] [28].

3.1.3. The Double-write Type

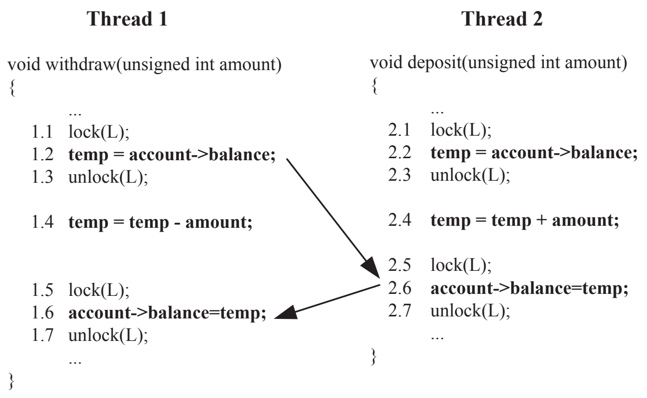
The double-write type is an inter-boundary vulnerability type that usually causes information leakage. A domain (primarily a high privilege domain) can write sensitive information to the other domain but later overwrites it with legitimate data in the same context scope. Such a situation can cause sensitive information disclosure as the receiver domain can obtain the initially written data within a limited duration.

In practice, a typical scenario of this kind is when a data structure containing pointers is used to store data both in the kernel mode and user mode. When a syscall copies such an internal kernel structure to the user space, it first copies the object to the memory, then, it adjusts the pointers to point to the corresponding userland buffers. However, since the original kernel-mode pointers reside in the user memory for a short period before being overwritten with the appropriate addresses, the user can use it to obtain sensitive kernel information and attack the kernel [28].

Figure 9 shows an example of the double-write type inter-boundary vulnerability that crosses the kernel-user boundary [2]. The structure `_USERNAME` is an object that includes a

[1]Linux use copy functions to manipulate pointers when crossing the kernel boundary, such as copy_from_user().

[2]Windows Kernel ring-0 address leak via a double-write in NtQueryVirtualMemory(MemoryMappedFilenameInformation). `https://bugs.chromium.org/p/project-zero/issues/detail?id=1456`



```
Thread 1

void withdraw(unsigned int amount)
{
        ...
    1.1 lock(L);
    1.2 temp = account->balance;
    1.3 unlock(L);

    1.4 temp = temp - amount;


    1.5 lock(L);
    1.6 account->balance=temp;
    1.7 unlock(L);
        ...
}
```

```
Thread 2

void deposit(unsigned int amount)
{
        ...
    2.1 lock(L);
    2.2 temp = account->balance;
    2.3 unlock(L);

    2.4 temp = temp + amount;

    2.5 lock(L);
    2.6 account->balance=temp;
    2.7 unlock(L);
        ...
}
```

Figure 10: A write-after-read type atomicity-violation bug in the thread boundary.

string and its corresponding textual buffer. A local object of this type is first initialized at lines 11-12 and later copied to the user at line 14. Since the pointer `Buf` in the object that passed to user-mode still contains a kernel address, it is overwritten with a pointer to the user buffer at line 15. However, a time window is available for another thread to capture the disclosed kernel pointer between lines 14 and 15, causing information leakage.

Owing to reasons such as the C language nature, the current compilation techniques, the system allocator design, and the code optimization schemes, unintentional information leakage is difficult to avoid in the interaction that crosses a boundary [28]. Such vulnerability can be used to leak any sensitive data type, especially the kernel-mode pointers, which could help the attackers to bypass KASLR (Kernel Address Space Layout Randomization). Besides, the disclosure of the kernel-mode pointers also facilitates the exploitation of other kernel vulnerabilities.

3.1.4. The Write-after-read Type

A write-after-read type is different from the previous two types. In this type, the local write relies on a value from the preceding local read that is then overwritten by the remote write. Although we haven't found real examples of this type in the hard boundaries, it is well-known in the thread boundary as a kind of atomicity-violation bug. Figure 10 shows an example of the write-after-read bug type that crosses the thread boundary. In this case, two threads are handling a shared bank account, one to withdraw the money (Thread 1) and the other to deposit the money (Thread 2). A shared variable `account->balance` is accessible to both of the two threads. An atomicity violation occurs when the read (line 1.2) and update (line 1.6) of `account->balance` in Thread 1 is interleaved by the update operation (line 2.6) in a concurrent running Thread 2 (as the arrows indicate). This atomicity violation makes the deposit operations in Thread 2 ineffective because the update (line 1.6) of `account->balance` in Thread 1 is based on the read (line 1.2) of `account->balance` in Thread 1, which occurs before the update (line 2.6) in Thread 2. Thus, the update of Thread 2 is overwritten by Thread 1. Similarly, the update operation in Thread 1 (line 1.6) can also interleave between the read (line 2.2) and update (line 2.6) of `account->balance` in Thread 2 to make the withdraw operations in Thread 1 ineffective.

The write-after-read type is rare in the hard boundary, and this is because of the privilege gap. Even if a remote write from a less privileged domain violates the write-after-read dependency in a privileged domain, the worst result is that the privileged domain writes wrong data to the less privileged one. However, this leaves no harmful result to the privileged domain.

3.2. Detection

Current detection research of the inter-boundary vulnerabilities mainly focuses on the kernel-user boundary. A few works also attempt to dig the hardware-OS boundary and the VMM-guest OS boundary.

3.2.1. The Kernel-user Boundary

For the kernel-user boundary, the research focuses on the double-read type inter-boundary vulnerability, also known as the double-fetch vulnerability. The detection approach generally has two categories: the dynamic approach and the static approach.

Jurczyk and Coldwind [23] carried out the first study on the double-fetch vulnerabilities in their Bochspwn project. They dynamically instrumented the Windows kernel in an emulator to observe read operations on the same user space address within a short time. They found 36 real EoP vulnerabilities from Windows. However, their analysis and findings are limited to Windows, and their dynamic approach has low code coverage and high runtime overhead. Schwarz *et al.* [20] combined the side channel cache-attack and kernel-fuzzing to detect double-fetch vulnerabilities. However, their approach is limited to Linux syscalls, missing significant numbers of vulnerabilities occur in non-syscall functions, such as drivers. Pan *et al.* [29] detect double-fetch vulnerabilities with a virtualization monitor. They capture various dynamic behaviors of kernel execution from the hypervisor level and detect kernel vulnerability based on these context information, which has a significant efficiency improvement over Bochspwn. However, as a dynamic approach, their approach still has a low path coverage. It can only test one execution path per time and cannot test drivers when the hardware is absent.

Wang *et al.* [16] presented the first static approach to detect double-fetch vulnerabilities in the Linux kernel. Using a pattern-matching analysis, they identified 6 new double-fetch vulnerabilities. Their approach can cover the complete kernel source code, including all drivers and all hardware architectures, without relying on the hardware support. However, their approach is limited to intra-procedure analysis, and the analysis partially relies on the manual review. Xu *et al.* [19] proposed a formal definition for the double-fetch bugs and used static analysis techniques with symbolic checking to vet for double-fetch bugs. Their approach can do inter-procedure analysis, which improves the bug coverage. However, their definition includes situations that are not currently buggy and only have the potential to turn into bugs when the code is updated. Besides, their approach needs to compile the source code to LLVM IR and specify the target architecture. Thus, it detects only one architecture at one time and misses the true case (such as CVE-2016-6130) when the source code cannot be compiled. Nevertheless, they identified a new double-fetch vulnerability (CVE-2017-15037) from FreeBSD. Wang *et al*. [30] leverage the fact that a double-fetch vulnerability lacks a recheck and proposed a static approach. Their approach first infers the "checks" in the branch based on the error code. Then, it employs backward analysis and data-flow analysis to find uses of the critical variables. Finally, it traverses execution paths to find potential modifications of the critical variables and check the absent rechecks. Their approach emphasized the use of the critical variables, which avoids the inaccuracy in Xu's [19] definition. However, it still has a false positive rate of 99.3%, and the 2,808 reported cases need manual confirmation.

In 2018, Jurczyk [28] reloaded the Bochspwn project and combined taint tracking to detect kernel information leak vulnerabilities, which includes the read-after-write type and the double-write type inter-boundary vulnerability. In addition, taint tracking is also leveraged by Wang *et al*. [18] and Wang *et al*. [30] to detect double-read type inter-boundary vulnerabilities.

3.2.2. The Hardware-OS Boundary

Lu *et al*. [25] presents the first dedicated study of the hardware double-fetch vulnerabilities between the peripheral hardware and the operating system, which provides a new perspective to the inter-boundary vulnerability by increasing the scope to include the hardware-OS boundary. Based on a static pattern-matching approach, they discovered four previously unknown double-fetch vulnerabilities in the hardware-OS boundary from the Linux kernel. However, their approach is preliminary, which has a high false report rate and relies on manual efforts. Then, Song *et al*. [31] used fuzzing to detect hardware double-fetch vulnerabilities in the hardware-OS boundary. They proposed a probing framework, which hooks into the kernel's page fault handling mechanism to monitor and log traffic between device drivers and their corresponding hardware, and mutate the data stream on-the-fly using a fuzzing component. By imitating the adversarial attacker on the peripheral devices, the framework found 9 previously unknown vulnerabilities. However, this approach cannot guarantee code coverage. Besides, crashes in the kernel lead to system reboots, which introduces significant latency. Thus, the efficiency is limited.

3.2.3. The VMM-guest OS Boundary

Wilhelm [1] used an approach similar to the Bochspwn project to detect double-fetch vulnerabilities in the VMM-guest OS boundary. He analyzed the memory access pattern of para-virtualized devices' backend components and discovered three new double-fetch vulnerabilities in the security-critical backend components. It is worth mentioning that one of the discovered vulnerabilities did not exist in the source code but was introduced through compiler optimization (CVE-2015-8550) because the compiler optimizes the code in a way that a second fetch is conducted instead of reusing the value from the first fetch. Wilhelm's work enriches the inter-boundary vulnerability diversity by including the VMM-guest OS boundary. However, his approach has similar drawbacks as Bochspwn's, such as low path coverage, high runtime overhead, and hardware support dependency.

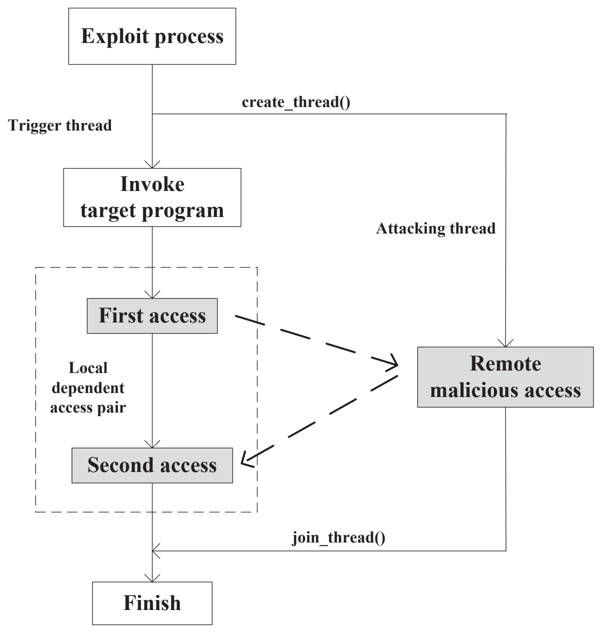


Figure 11: The process of the inter-boundary vulnerability exploitation.

3.3. Exploitation

The key to exploiting an inter-boundary vulnerability is to insert the remote less privileged access to interleave between the privileged dependent access pair, shown in Figure 11. Different from the exploitation of the concurrency errors that only need to switch the thread scheduling, to exploit the inter-boundary vulnerability, the attacker has to create a remote attack thread to perform the malicious behavior. When the remote access is a write, the goal is to tamper the shared data to violate the dependency between the local access pair, causing memory corruption. When the remote is a read, the goal is to read the shared data between the two local writes, causing information disclosure. For the former, shared data tampering can occur in several situations. (1) Race conditions. The shared data structures (e.g., global variables and heap objects) can be modified under races, and shared variables between threads can also be modified under races. (2) Design errors. The shared data can be potentially modified by the thread itself owing to the logic errors in the design. (3) Implementation defect. Shared data can also be modified by a vulnerable implementation, such as type casting or integer overflow [30]. (4) Hardware data flip. For vulnerabilities in the hardware-OS boundary, the shared data in the I/O memory can be flipped from the hardware side.

As far as we have investigated, currently, viable exploitation of the inter-boundary vulnerability is limited to the double-read type. The exploitation process involves two threads: a triggering thread to invoke the vulnerable function, and a racing thread to flip the data within the time window of the two fetches [17]. Jurczyk and Coldwind also gave the first study on exploiting the double-fetch vulnerability. They pointed out that the key to successfully exploit a double-fetch vulnerability is to broaden the time window between the two fetches. They provided some tricks to realize this goal, such as placing the shared data across the page boundary, disabling the page cacheability, and flushing the TLB (Translation Lookaside Buffers) before reading the data [23].

Hammou [32] achieved privilege escalation by exploiting a double-fetch vulnerability in a Windows driver. The kernel-mode driver fetches a user pointer `UserAddress` twice, the first fetch accesses the IRP SystemBuffer to get the address (pointed by `UserAddress`) of the user input, and the second one performs `ProbeForWrite` on it to get the data. By supplying a valid user-mode address to `ProbeForWrite` for check and then quickly switching the `UserAddress` field to a desired kernel-mode address, the exploitation can write 4 bytes data to any kernel memory location. In this way, an attacker can set the `TokenObject->Privileges.Enabled` field of the attacker's process to 0xffffffff to escalates the privilege [17].

Similar exploitation work also includes CVE-2005-2490 and CVE-2016-6516. Maiki provided exploitation on CVE-2005-2490 and placed the crafted user data across the page boundary, which could increase the time window by triggering a page fault when accessing the data from more than one page [33]. A page fault could suspend the working thread that is fetching the user data. Meanwhile, a malicious thread could be swapped in and tamper the data before the fetching thread gets scheduled again [17]. Bauer [34] exploits CVE-2016-6516 and could control which cache the overflow happens on [17]. However, the success rate is limited as the time window is usually very small. Schwarz *et al.* [20] proposed an exploitation approach based on the flush + reload side-channel attack, which could precisely trigger the data switch right after the first fetch, increasing the success rate to 97%.

Theoretically, the exploitation process of the read-after-write type is basically the same as the double-read type except that the remote data tampering is inserted to a local access pair whose first access is a write. For the double-write type, reading data within the time window is more workable than writing data to it. Besides, failure of reading data within the time window causes no effect, however, writing data outside the time window causes an unpredictable result. Thus, the exploitation of the double-write types is more accessible than the other types.

3.4. Prevention

Known attempts to prevent inter-boundary vulnerabilities are also limited to the double-read type. Based on the investigation of the real-world vulnerability patches, Wang *et al.* [16] provided strategies on preventing the double-fetch vulnerabilities and invented a tool based on pattern-matching to patch known vulnerabilities automatically. It was the first attempt to prevent and fix such vulnerabilities. Their strategies are summarized as the following suggestions.

- **Copy the data incrementally**. For the data structure that has a double-fetched part, we only read that part once, and skip that part to read the rest for the second time [16].
- **Use the data from the same read**. Even though some data is read twice, we only use the version from the first read and ignore the second read, which avoids the risk of being tampered [16].
- **Override with values from the first read**. To avoid the cross-use of data from different reads, we use the data from the first read to override the data from the second read. Thus, only one data version is left and used [16].
- **Check the data before use**. Compare the data from the first read with the data of the second read. If the data is not identical, the operation is safely aborted. This approach can both allow detecting attacks by malicious users and protecting from situations in which the data is changed without malicious intent, such as design errors and implementation defects [16].

The key to these strategies is to avoid using both data from the two reads. Although their prevention strategies only give suggestions on how to avoid such vulnerability and rely on the experience of the programmer, they are classic and influence the followers. Xu *et al.* [19] also provide similar but refined suggestions to mitigate such vulnerabilities. However, such prevention strategies are only limited to the double-read type and not suitable for other types.

In addition to these strategies, Schwarz *et al.* proposed to eliminate double-fetch vulnerabilities with hardware transactional memory [20] (e.g., Intel TSX and ARM TrustZone), which is innovative and effective. In addition to the manifested vulnerabilities, their approach can also eliminate the compiler introduced one (CVE-2015-8550). Luo *et al.* proposed to combine pattern-matching and transactional memory to prevent such vulnerability [35]. A merit of such transactional memory-based approach is that it is suitable for all inter-boundary vulnerability types.

Table 2
Statistics of the known vulnerability investigation

Boundaries	R-R	W-R	W-W	R-W	Sum
Kernel-User	100	1	3	-	104
Hardware-OS	8	-	-	-	8
VMM-guest OS	3	-	-	-	3
Total	111	1	3	-	115

4. In-depth analysis

4.1. Statistics

We investigated 115 known inter-boundary vulnerabilities collected from academic works [36, 1, 17, 28] and the CVE (Common Vulnerabilities and Exposures) database. For some of the vulnerabilities, we also tried to obtain the buggy source code and corresponding patches from the relevant repositories and archives, which includes the Linux repository on Github, the Linux Kernel Mailing List Archive, and the Kernel Bugzilla. We have made the vulnerabilities we collected available for the security community for further research[3].

As Table 2 shows, we categorize the vulnerabilities we collected by the vulnerability type and the boundary. From the perspective of the boundary, most of the investigated vulnerabilities are subject to the kernel-user boundary, accounting for 90.4%. This percentage is following the fact that most research focuses on the interaction between the kernel and user space. From the perspective of the vulnerability type, the double-read type is the most prevalent type that manifests in all the three boundaries, which has an overall proportion of 96.5%. On the contrary, we did not identify any write-after-read type from the known vulnerabilities. This is because of the privilege gap. For a write-after-read type, even if a remote write from the less privilege domain violates the write-after-read dependency in a privileged domain, the less privileged domain cannot corrupt the privileged one. However, this type is not rare in the concurrency bugs, whose communication partners have an equal privilege. For the read-after-write type and the double-write type, though only manifested in the kernel-user boundary, theoretically, they could also exist in the other two boundaries. Take the double-write type as an example. In addition to the three disclosed vulnerabilities in the kernel-user boundary, the researcher also found the other 20 kernel pool address leakage vulnerabilities that are subject to the double-write type [37]. However, since Microsoft changed the bar for a security bulletin, their reports were instead targeted to be fixed in the next version of Windows. Unfortunately, these reports were not revealed. For the read-after-write type, Lu *et al*. [38] also found plenty of such cases when detecting double-read type vulnerabilities in the hardware-os boundary. However, they did not give a detailed analysis at that time and left them to future work. Thus, in the future, we should pay more attention to these two vulnerability types, especially in the unmanifested boundaries.

4.2. Consequences

According to the real-world case analysis, inter-boundary vulnerabilities can cause two kinds of direct consequences: memory corruption and information disclosure. Both of the double-read type and the read-after-write type can cause memory corruption, which accounts for 97.4%, while the double-write type can only cause information disclosure, which has a proportion of 2.6%. More specifically, memory corruption can further cause denial-of-service, privilege escalation, auditing bypass, and also information disclosure. It is worth noting that denial-of-services have various reasons, such as buffer overflow, array over access, and NULL pointer dereference. Besides, though the memory corruption can also lead to information disclosure, it is different from the one caused by the double-write type. The double-write type discloses the intermediate sensitive data within a short duration, while the disclosure in the memory corruption is usually caused by buffer over-bound write to the user space. As Figure 12 shows, the overall proportions of the major consequences of the investigated cases are 22%, 39%, 32%, and 7% for denial-of-service, privilege escalation, auditing bypass, and information disclosure, respectively.

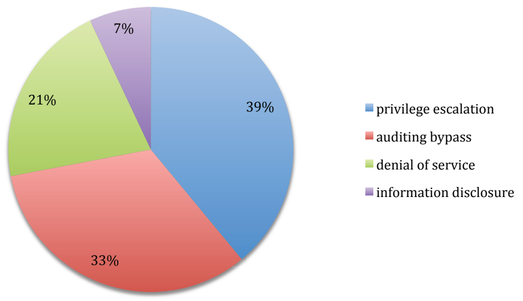


Figure 12: The percentages of different consequences.

4.3. Differences

Though concurrency plays a significant role in the root cause of inter-boundary vulnerabilities, the inter-boundary vulnerability is different from the concurrency bugs. The atomicity-violation bug is one of the most common and significant concurrency bug types. An atomicity violation occurs when a code block is unexpectedly interleaved by operations from other concurrent threads. It turns to an atomicity-violation bug if the unfortunate interleaving breaks the atomicity assumptions made by the programmer and leads to incorrect program behaviors. Atomicity-violation bugs widely exist because many programmers are used to sequential thinking and frequently assume code regions to be atomic without proper enforcement [9]. However, the inter-boundary vulnerability is different from the atomicity-violation bug in the following aspects.

- **Different boundaries.** The concurrency bugs occur among regular threads. However, the inter-boundary vulnerabilities exist in different system boundaries, such as the kernel-user boundary, the hardware-OS boundary, and the VMM-guest OS boundary.

- **Different privilege levels.** The concurrency bugs occur in the communication between domains that have an equal privilege level. In contrast, the inter-boundary vulnerability has a privilege gap between the communication partners, which can cause more severe consequences, such as privilege escalation.

- **Different research focuses.** Research on the concurrency bugs focuses on waving the thread interleaving

[3]https://github.com/wpengfei/IBV.git

to discover the misuse of synchronization primitives, which does not take the existence of a malicious communication partner into account. However, the study of the inter-boundary vulnerability focuses on safe communication over the trust boundary, including potential malicious behaviors.

A Time-Of-Check to Time-Of-Use (TOCTTOU in short) bug occurs when a program checks for a particular characteristic of an object, to take some action based on the assumption that the characteristic still holds. However, it actually does not hold any longer [39]. TOCTTOU bugs are most common in the Unix file system, which can date back to the 1990s. A TOCTTOU bug is similar to the double-read inter-boundary vulnerability, and the data inconsistency in a TOCTOU bug is usually caused by a race condition owing to improper synchronization of the concurrent accesses to a shared object. However, the TOCTTOU bug is specific to the shared objects (e.g., a file or a socket), and it usually does not cross a privilege gap, which is different from the inter-boundary vulnerabilities.

Thus, the inter-boundary vulnerabilities are different from the well-studied concurrency bugs and TOCTTOU bugs, and worth a dedicated study.

5. Perspective

In this paper, we choose the kernel-user boundary, the hardware-OS boundary, and the VMM-guest OS boundary as representatives to illustrate the inter-boundary vulnerability. However, in addition to these three boundary types, other boundaries in the system can also cause inter-boundary vulnerabilities. For instance, the wrapped interfaces, such as the syscall wrapper functions, handle parameters from the user application and introduce a new boundary between the syscall and the user application [40]. Besides, technologies also create new system layers and form new boundaries, such as the container in Docker forms a boundary between the container and the application, which also has a privilege gap and thus could cause inter-boundary vulnerabilities. Future work should bring more such boundary types into research. From Table 2, we can see that, compared to the double-read type, the read-after-write type and the double-write type are less aware and lack systematic study. Future work should also pay more attention to these two vulnerability types, especially in the unmanifested boundaries.

So far, the research to the inter-boundary vulnerabilities concentrates on the detection, while the prevention and fix works are minimal and preliminary. The inter-boundary vulnerability is difficult to detect, which can hide for over ten years (CVE-2016-6480) before being exposed [16]. Since the malicious behaviour in the communication partner is potential, state-of-the-art fuzzing-based tools [41, 42, 43, 44] or concurrency bug tools [9, 10, 11, 12, 13, 14] are not workable to detect such vulnerability. Thus, researchers take advantage of the specific patterns of the inter-boundary vulnerability to devise dedicated tools. Based on the investigation of the source code, Wang *et al*. [16] extract three typical patterns that double-fetch vulnerabilities are prone to occur, including the size-checking, type-selection, and shallow copy. Then, Xu *et al*. [19] provided more refined patterns, including the dependency lookup, protocol/signature checking, and information guessing. For the hardware double-fetch vulnerability in the hardware-OS boundary, Lu *et al*. [25] also categorized patterns based on the feature of the I/O memory, including common check, loop check, wait check, stable check, configure check, check and use, and block check. These patterns are beneficial for improving the accuracy of the detection approach. However, there are obvious shortcomings of current approaches. Dynamic approaches have limited code coverage, high runtime overhead, and rely on hardware support; static approaches have high false positives, rely on the source code, and to some extent, require additional manual efforts. Future work should try to overcome these challenges.

During the real-world case analysis, we noticed some individual inter-boundary vulnerability cases caused by faults at the implementation level instead of the commonly-seen coding phase. For example, the case (CVE-2015-8550) introduced by compiler optimization [1] and the case occurred in a macro [17]. Such cases are less-aware but more destructive. Moreover, some cases are platform-specific. For example, the read-after-write situations are more likely to cause vulnerabilities at the Windows platform because it uses direct pointer dereference to access inter-boundary data, while other platforms, such as Linux and FreeBSD, use dedicated wrapper functions. Lacking a clear distinction between different domain pointers in Windows makes it bug-prone. Future work should also consider such implementation and platform feature to dig hidden cases.

For prevention, known works mainly provide suggestions to improve developers' coding habits to mitigate the occurrence of the vulnerability. However, such mitigation strategies are only limited to the logic of the double-read type. From the perspective of system survivability, sometimes, tolerate the attacks caused by such inter-boundary vulnerabilities is more practical and viable than discovering and fixing the vulnerability. As an open problem, improve the system's robustness is another solution for the system designer. One attempt is using the hardware transactional memory (e.g., Intel TSX and ARM TrustZone) to keep the atomicity of the accesses from the same domain [20][35], which is theoretically helpful to all the inter-boundary vulnerability types. However, it depends on hardware support. Another possible way is memory read-only mapping. Modern operating systems nowadays implement read-only memory mappings at their CPU architecture level to prevent security attacks. By mapping memories as read-only, the kernel can trust the memory content, eliminating the potentially malicious data mutation from the user space. Thus, it is useful for the resistance of double-read and read-after-write type vulnerabilities. However, a significant drawback is that not all the user data can be mapped as read-only. In the future, we should lay more emphasis on approaches to practically prevent or tolerate such vulnerabilities.

The study of the inter-boundary vulnerability also shows insights for some open problems in system security. For example, "do the inter-boundary patterns also motivate other vulnerabilities?", "can such problem occur in a different level of granularity throughout the system?", "is there an unexpected way to exploit such vulnerability?" At the end of this paper, we would like to provide a preliminary answer.

Most of the double-read type inter-boundary vulnerabilities we investigated are the "check-use" situation, where the check comes first and followed by the use, and the atomicity of the "check-use" relation is violated during inter-boundary memory accesses. However, a particular situation exists in the system where the shared data is used before checking, i.e., a "use-check" situation, which can also lead to a security vulnerability. This is because, to optimize the instruction processing performance, modern CPUs adopt out-of-order execution and speculative execution. However, these mechanisms allow the CPU to prefetching sensitive memory data to the CPU cache before verifying the privilege, breaking the privilege isolation of the system, which can lead to severe vulnerability such as Meltdown [45] and Spectre [46]. These vulnerabilities occur at the instruction level, which is a finer granularity than regular inter-boundary vulnerabilities. An attacker can obtain sensitive information in the CPU cache via a "flush+reload" side-channel attack. Although side-channel is not a conventional way to exploit the inter-boundary vulnerability, it has been used by Schwarz *et al*. [20] to exploit the double-fetch bug. Since the discovery of Meltdown and Spectre was a piece of breaking news in 2018, and these vulnerabilities have been well-studied, we would not repeat any longer. We hope the above discussion could give some inspiration for future research.

6. Conclusions

The inter-boundary vulnerability is a new research point that worth dedicated study. In this paper, we gave the first in-depth study of it. We investigated three commonly seen boundaries in the system that inter-boundary vulnerabilities are prone to occur. We extracted four inter-boundary vulnerability types based on the investigation of 115 real-world cases and illustrated the principle of each type. We also discussed the state-of-the-art techniques that are relevant to the detection, prevention, and exploitation of such vulnerabilities, aiming to light the future research on this topic.

Acknowledgement

The authors would like to sincerely thank all the reviewers for your time and expertise on this paper. Your insightful comments help us improve this work. This work is partially supported by the National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013), the HUNAN Province Natural Science Foundation (2017RS3045, 2019JJ50729), and the National Natural Science Foundation China (61472437, 61902412, 61902416).

References

[1] Felix Wilhelm. Tracing privileged memory accesses to discover software vulnerabilities. Master's thesis, Karlsruher Institut für Technologie, 2015.

[2] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214. ACM, 2007.

[3] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. Locksmith: Practical static race detection for c. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(1):3, 2011.

[4] Jun Chen and Steve MacDonald. Towards a better collaboration of static and dynamic analyses for testing concurrent programs. In *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*, page 8. ACM, 2008.

[5] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[6] Koushik Sen. Race directed random testing of concurrent programs. *ACM SIGPLAN Notices*, 43(6):11–21, 2008.

[7] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: crowdsourced data race detection. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 406–422. ACM, 2013.

[8] Kai Lu, Zhendong Wu, Xiaoping Wang, Chen Chen, and Xu Zhou. Racechecker: efficient identification of harmful data races. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 78–85. IEEE, 2015.

[9] Shan Lu, Soyeon Park, and Yuanyuan Zhou. Finding atomicity-violation bugs through unserializable interleaving testing. *IEEE Transactions on Software Engineering*, 38(4), 2012.

[10] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems*, 2006.

[11] Min Xu, Rastislav Bodík, and Mark D Hill. A serializability violation detector for shared-memory server programs. In *Programming Language Design and Implementation*, 2005.

[12] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Symposium on Principles of Programming Languages*, 2004.

[13] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *Object Oriented Programming Systems Languages and Applications*, 2012.

[14] Jeff Huang and Charles Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 144–154. ACM, 2011.

[15] Pengfei Wang, Jens Krinke, Xu Zhou, and Kai Lu. Avpredictor: Comprehensive prediction and detection of atomicity violations. *Concurrency and Computation: Practice and Experience*, page e5160, 2019.

[16] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1–16, 2017.

[17] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. A survey of the double-fetch vulnerabilities. *Concurrency and Computation: Practice and Experience*, 30(6):e4345, 2018.

[18] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. Dftracker: detecting double-fetch bugs by multi-taint parallel tracking. *Frontiers of Computer Science*, 13(2):247–263, 2019.

[19] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678. IEEE, 2018.

[20] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated de-

tection, exploitation, and elimination of double-fetch bugs using modern cpu features. *arXiv preprint arXiv:1711.01254*, 2017.

[21] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. Unix network programming, volume 2. In *Addison-Wesley Professional*, 2004.

[22] Fermin J. Serna. MS08-061 : the case of the kernel mode double-fetch. [Online.] https://blogs.technet.microsoft.com/srd/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/.

[23] Mateusz Jurczyk and Gynvael Coldwind. Identifying and exploiting windows kernel race conditions via memory access patterns. Technical report, Google Research, 2013. [Online]. http://research.google.com/pubs/archive/42189.pdf.

[24] Matousec. Khobe-8.0 earthquake for windows desktop security software. [Online]. http://www.matousec.com/info/articles/khobe-8.0-earthquake-for-windows-desktop-security-software.php, 2010.

[25] Lu Kai, Peng Fei Wang, Gen Li, and Zhou Xu. Untrusted hardware causes double-fetch problems in the i/o memory. *Journal of Computer Science and Technology*, 33(3):587–602, 2018.

[26] Andrew S. Tanenbaum. *Modern Operating Systems*. 2002.

[27] Paul Barham, Boris Dragovic, Keir Fraser, H Steven, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, volume 37, 2003.

[28] Mateusz Jurczyk. Detecting kernel memory disclosure with x86 emulation and taint tracking. [Online]. https://j00ru.vexillium.org/papers/2018/bochspwn_reloaded.pdf, 2018.

[29] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *26th USENIX Security Symposium*, pages 149–165. USENIX Association, 2017.

[30] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1899–1913. ACM, 2018.

[31] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary.

[32] Souhail Hammou. Exploiting windows drivers: Double-fetch race condition vulnerability. [Online]. http://resources.infosecinstitute.com/exploiting-windows-drivers-double-fetch-race-condition-vulnerability/, 2016.

[33] Maiki. Vulnerability caused by inconsistency checking. [Online]. https://maikiforever.wordpress.com/2011/10/07/, 2011.

[34] Scott Bauer. Linux >= 4.5 double fetch leading to heap overflow. [Online]. http://www.openwall.com/lists/oss-security/2016/07/31/6, 2016.

[35] Y. Luo, P. Wang, X. Zhou, and K. Lu. Dftinker: Detecting and fixing double-fetch bugs in an automated way. In *Wireless Algorithms, Systems, and Applications*, 2018.

[36] Mateusz Jurczyk and Gynvael Coldwind. Bochspwn: Identifying 0-days via system-wide memory access pattern analysis. Black Hat 2013, 2013. [Online]. http://vexillium.org/dl.php?BH2013_Mateusz_Jurczyk_Gynvael_Coldwind.pdf.

[37] TinySec. I am also got multi case of "double-write". [Online]. https://twitter.com/TinySecEx/status/943410888119218176, 2017.

[38] Kai Lu, Peng-Fei Wang, Gen Li, and Xu Zhou. Untrusted hardware causes double-fetch problems in the i/o memory. *Journal of Computer Science and Technology*, 33(3):587–602, 2018.

[39] Matt Bishop, Michael Dilger, et al. Checking for race conditions in file accesses. *Computing systems*, 2(2):131–152, 1996.

[40] Robert NM Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *First USENIX Workshop on Offensive Technologies*, WOOT '07, 2007.

[41] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2017.

[42] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358. ACM, 2017.

[43] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. *Proceedings of the IEEE PressSymposium on Security and Privacy*, pages 279–293, 2019.

[44] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 729–743, 2018.

[45] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[46] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

Pengfei Wang received his B.S., M.S. degrees, and Ph.D. in 2011, 2013, and 2018, respectively, from the College of Computer, National University of Defense Technology, Changsha. He is now an assistant professor in the College of Computer, National University of Defense Technology. His research interests include operating systems and software testing.

Xu Zhou received his B.S., M.S. degrees, and Ph.D. in 2007, 2009, and 2014, respectively, from the College of Computer, National University of Defense Technology, Changsha. He is now an assistant professor in the College of Computer, National University of Defense Technology. His research interests include operating systems and parallel computing.

Kai Lu received his B.S. degree and Ph.D. in 1995 and 1999, respectively, from the College of Computer, National University of Defense Technology, Changsha. He is now a professor in the College of Computer, National University of Defense Technology. His research interests include operating systems, parallel computing, and security.