Seminar Paper: A look at Bochspwn - Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns

Benjamin Halbrock

Abstract

Bochspwn by Mateusz Jurczyk and Gynvael Coldwind uses the Bochs X86 Emulator [1] to identify race conditions at the userland kernel boundary by analysing runtime memory access patterns. The paper analyzes the Windows NT kernel, identifies multiple candidates of such race conditions and discusses techniques for a successful exploitation, like a way to widen the attack window.

1 Introduction

As computers become increasingly prevalent, much effort is spend on securing them. Of big concern is not only the hardening of user applications but also finding and fixing bugs within the kernel of the operating system and its interfaces to userland.

Bochspwn is an instrumentation module for the Bochs X86 Emulator which is used to find a special case of race conditions between kernel and userland threads called double fetches. To do so it analyzes memory access patterns at runtime.

These double fetches can result in information disclosure of kernel data, a denial of service attack, or a successful local privilege escalation and therefore a possible sandbox breakout [6, p. 4f, 48f, 65]. This predicate does not only apply on multi-core systems, but is also true for single-core systems [8, p. 6].

As the testing of the yet unproven technique of using memory access patterns to identify double fetches [8, p. 5] called for fast iterations, the authors chose the Bochs X86 Emulator as the basis for their implementation. Bochs as an open source project could, as expected, easily be modified to fit their needs [6, p. 11]. The result of their efforts is called Bochspwn, an instrumentation module for Bochs, and has identified over 80 potential bugs within the Windows kernel [4, p. 23].

The following section gives some background information about synchronisation when using shared memory and explains the time-of-check-to-time-of-use race condition together with an introduction into exploitation techniques for this kind of bug. Section 3 focuses on the approach taken by Mateusz Jurczyk and Gynvael Coldwind and explains the major design decisions, along with remarks regarding the paper. It is followed by a discussion of techniques to widen the window of opportunity for an successful attack. The paper finishes by giving an overview of the consequences the initial publication had on the research community and a short conclusion.

2 Background

This section explains the technical circumstances in greater detail. The idea of **shared memory** as a technique for communication between threads is introduced first. Then the **Windows address space** layout is explained, followed by an explanation of the problem of synchronisation when using shared memory on the example of **double fetches**. Lastly two techniques for **(ab)using the race condition** bugs under Windows get explained.

2.1 Shared Memory

It is very common to share data within an operating system by mapping memory segments in the address spaces of different threads. The main advantage of this technique is that once the mapping has been established there is no additional overhead when writing or reading data, as each thread can freely read and write its memory segment. Those memory segments can additionally be used between threads with different access rights and are used for some communication between threads in userland and kernel routines in Windows. When data is transferred via a shared memory segment, special care must be taken to synchronise concurrent accesses, as by default there is no such mechanism.

2.2 Windows address space

The Windows kernel separates the address space in two parts. A system wide kernel address space and many (mostly one per thread) user address space regions, which can be accessed in user- and kernel-mode. The boundary is set at the address 0x8000 0000 and visualised in Figure 1. For a thread in kernel-mode the userland portion of the

address space is freely read- and writeable. By having a pointer at it, any data is basically an implicitly shared resource for a kernel-mode thread [6, p. 2]. Because of this fact, a user-mode thread can share data with a kernel thread by simply passing a pointer.

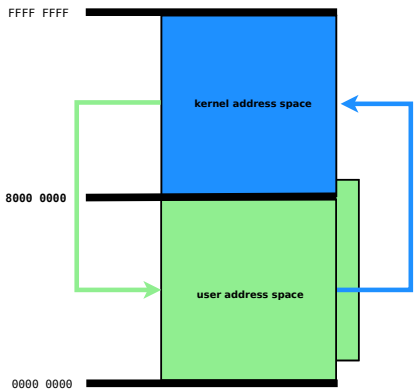

Figure 1: Address space seperation in 32bit Windows

As a Windows kernel thread can access the global kernel data and the data of the user thread without special checks, a pointer into the userland portion of the address space can be modified to point into the kernel address space by flipping bit 31 (XOR 0x8000 0000).

2.3 Double Fetches

Even when there are synchronisation mechanisms in place, a communication partner can simply chose not to participate [8, p. 4]. To eliminate this problem and to maintain consistency of userland data while in kernel mode a kernel routine normally copies the needed data into a local buffer [6, p. 2].

If the kernel routine fetches the underlying value multiple times, for example when first checking the size of the given value and a second fetch for copying the data, all the conditions for a special kind of race condition are met. These types of race conditions are called time-of-check-to-time-of-use race conditions or double fetches in short [7].

Double fetch means that the value read on the first access does not have to match the value of the following fetches, thus breaking the assumption of the developer of it staying the same. This broken assumption can have undesirable effects when a kernel function firstly checks the value, which is then immediately changed by an application in userland, and finally the changed and now unchecked value gets used in the kernel function. This unchecked value can now directly influence kernel memory, which should be unread- and unchangeable by applications in user-space. Take a simple copy operation as an example. The userland process calls the kernel function with a pointer to the data and a pointer to the size of the data. The kernel function then uses the given size to check whether the data fits in an allocated buffer. Directly after this check, the data should be copied based on the previously checked size. But in the instance of a double fetch bug, the size gets fetched twice. Once for the check and a second time for the actual copy operation. By double fetching, the value can be different on the second fetch and the copy operation, which is running in kernel-mode, can now read or write out of the previously checked bounds.

2.4 Exploiting the race condition

A simple exploit uses two threads, one executing the syscall (the racing thread), and a different one, the flipping thread, trying to change the shared value at the right moment. The right moment for altering the shared data is after all checks are performed and before the data is copied in a local and secure buffer.

The raced value can be:

- a pointer to some value in memory
- a buffer size, a so called arithmetic race (shown in Listing 1)

If the shared value is given by a pointer, the attacker can use the XOR instruction on the bit 0x8000 0000. This instruction will flip the pointer between user- and kernel-space, as illustrated in Figure 1. Although such an attack is precise in the value it flips to, it is less likely to succeed than an arithmetic race, as it requires an uneven number of flips, leading to a maximum winning ratio of 50 percent. The other case is changing a given size value for a buffer, an arithmetic race. An example is given in Listing 1.

Listing 1: Example of an arithmetic race, where **BufferSize** is the raced value [5]

```
PDWORD BufferSize =
        /* user-mode address */;
PUCHAR BufferPtr =
        /* user-mode address */;
PUCHAR LocalBuffer;

LocalBuffer = ExAllocatePool(
        PagedPool, *BufferSize);
```

```
if (LocalBuffer != NULL) {
  RtlCopyMemory(LocalBuffer,
        BufferPtr, *BufferSize);
} else {
  // bail out
}
```

Winning an arithmetic race is much easier, as an attacker can add some value to the given buffer-size, resulting in data of a different size than checked being read or written by the kernel, possibly breaking boundaries. A shortcoming of the arithmetic races is that they are imprecise as it is not clear how often the flipping thread is modifying the given value [4, p. 11].

3 Approach

To identify possible double fetch candidates Jurczyk and Goldwind started by analysing known bugs and identified common traits of interesting memory accesses [6, p. 14ff] of time-of-check-to-time-of-use race conditions:

- the code executes in kernel mode
  this can lead to false positives in the recognition of double fetches, but as those mostly occur during the initialisation and boot process, they can be easily filtered out
- at least two reads from the same virtual address
  - **writeable from user space**
    as checking the actual mapping for every memory access would be very slow (traversing page tables), it is initially only checked whether the address is "within the user-mode virtual address space boundaries" [6, p. 16]. This check serves as a fast filter.
  - **within a short time frame**
    particularly within one syscall. This requires keeping a larger cache of memory reads and detecting *sysenter* and *sysexit* instructions. The actual detection code can afterwards use real timing information.

They decided to extend the Bochs X86 CPU emulator, as it can be easily modified, by using the provided hooks, and different strategies for analysing the system can be tested. A major downside of this approach is the slow speed that the analysed system is running at, as all instructions are emulated. The chosen approach is by design unable to detect shared memory communication between an untrusted process and a system- or administrator-owned process, or hypervisor mode double fetches [6, p. 16]. This restriction exists, because only code with an execution origin in kernel-mode is logged and inspected, leading to an additional slowdown of those sections. By analysing all memory accesses, using a tree or hash-map, it should be possible to also detect those double fetches, but this was outside the scope of the paper and would probably be even slower.

A faster technique is modifying the exception-handler of the operating system. This approach will not slow the system down very much, but requires patching the Operating System, which can be cumbersome. Another possibility is to use a hypervisor to detect double-fetches. This solution does also not result in a huge slowdown of the system, as most code can run natively and only interesting sections need to be inspected by the hypervisor. Although it is harder to implement F. Wilhelm did so in [8]. The approach of tracing memory accesses of shared pages proved to be faster, but less flexible, e.g. for collecting stack traces. Further possibilities would be doing a static code analysis, which is very hard when dealing with hardware drivers and low level instructions, or the usage of a hardware CPU debugger which is very expensive [3, minute 5ff].

Another inherent problem of a runtime analysis is code coverage. To test as many syscalls as possible the team was "running various additional applications within the guest (like programs making use of the DirectX API), as well as the Wine API unit-test suite" [6, p. 24], resulting in an instruction coverage of about 40% in the ntoskrnl.exe (the Windows kernel) and about 50% in win32k.sys (the kernel-mode device driver) [2] [3, minute 30ff]. Unfortunately the basis for those numbers is not clear.

The basic procedure to finding bugs using Bochspwn is [4, p. 20]:

- start an operating system within Bochs with Bochspwn enabled
- run interesting (invoking syscalls) programs like benchmarks or test-suites
- wait for all tests to finish and the OS shutdown
- process and filter the generated logfile

To identify double fetches Bochspwn implements an offline and online mode. Both are watching all memory accesses from kernel-mode into userspace (most significant bit in 32bit mode, most significant 16bits in 64bit mode) operating with ring-0 privileges [6, p. 15]. To find those accesses and differentiate between syscall handlers, Bochs is hooked to watch for *syscall* and *sysenter* instructions. Linear

memory accesses are additionally hooked to identify the physical address and perform checks to filter out uninteresting memory accesses. The offline mode writes a log of all those memory accesses along with debug information. This process generates a logfile that can easily reach a size of a hundred gigabytes [6, p. 23].The online mode on the other hand is writing only a small log of possible double-fetch candidates, by doing an online check against the known double-fetch access patterns, increasing the CPU and memory usage. The online-mode holds a cache of recent memory accesses of a thread for these checks, which gets evaluated for double fetch candidates on the next *sysenter* instruction.

Additional debug information like the *process id* and the *call stack* is gathered by parsing the kernel-memory structures of the emulated operating system. The information gathering slows the execution down considerably, is highly system specific and needs to be adjusted for every guest operating system. A Windows boot takes about 20 minutes in offline mode and about 35 minutes in online mode. It is relevant to note that the offline mode, while offering more flexibility in the analysis, requires extensive post processing like splitting the logfile, removing noise produced by some drivers and finding double fetches within the log. The post processing takes minutes per step and depends on the size of the logfile [6, p. 23]. The online mode does all of these checks at runtime and is approximately two times slower than the offline mode [6, p. 24].

The authors then manually validated double fetch candidates found this way and reported them to Microsoft.

3.1 Widening the window of opportunity

Mateusz Jurczyk and Gynvael Coldwind identified two classes of possible exploitation methods for a double fetch vulnerability and differentiated them by the raced value:

- a pointer to some value in memory (precise, but less likely to succeed)
- a buffer size, a so called arithmetic race (imprecise, but more likely to succeed)

As both of them get explained in Section 2.4, this section instead explains methods to increase the chance of winning such a race. One class of those techniques focuses on widening the opportunity for a successful exploitation by enlarging the time window in which the raced value can be flipped.

To show that even double fetches that are executed right after another can be a viable target for exploitation, the authors evaluated multiple techniques to extend the time window for exploitation and therefore the win ratio.

A very effective technique the authors found, is to split the shared value over two physical memory pages. The flipping thread will now flip the lower half of the address, only fetching one page, while the kernel function will have to fetch two pages. The optimal setup which leads to an five times increase of the execution time [4, p. 29] can be seen in Figure 2.

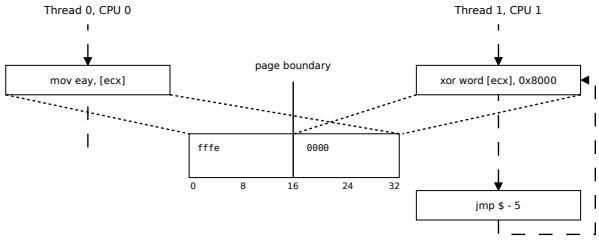

Figure 2: shared value crossing page-boundary [6, p. 31] on *little endian* (thread 0 is the kernel thread)

The parameters to choose for the individual pages are architecture dependant. According to their measurements the best solution for hyper-threading is to have one flipping and one racing thread share one physical core and to leave all caches enabled, so that the raced value can reside there and be rapidly flipped [4, p. 46].

When running on physical cores, disabling the caches for the racing thread is the way to go [6, p. 38ff]. Caches for a page can be disabled by a thread in user-mode by using *VirtualAlloc* with the `PAGE_NOCACHE` flag.

In some cases it is a good idea to flush the translation lookaside buffer (TLB) to slow down the kernel-mode thread. The subsequent page walk takes over 2500 cycles and can be achieved by calling *Virtual Unlock* on unlocked pages. This call is always cost effective [4, p. 41], as it costs less cycles to call than the overhead it introduces to the next fetch. A downside of this approach is that it has to be precisely timed, that only the kernel-mode thread will have to perform the page-table-walk and that it slows down the racing thread. Because of this slowdown, flushing the TLB will not always result in a higher win ratio [6, p. 34f].

3.2 Consequences

Mateusz Jurczyk and Gynvael Coldwind showed that a runtime analysis of memory access patterns is a viable way to identify time-of-check-to-time-of-use race conditions without needing to modify an existing operating system or special hardware. As emulating an X86 CPU comes at a big performance penalty, the authors mention working on a Virtual Machine Monitor (VMM) based solution called *HyperPwn* [6, p. 64]. Unfortunately it does not seem to have been fully implemented or released. Fortunately Felix Wilhelm successfully implemented a VMM based memory access pattern analyser in his master thesis *Tracing Privileged Memory Accesses to Discover Software Vulnerabilities* [8], showing that it can be done and is reasonably fast. It is interesting to note that some of the disclosed bugs were probably caused by a shared macro and that Microsoft did analyse and fix the root cause of these issues [6, p. 41, 58].

4 Conclusion

Mateusz Jurczyk and Gynvael Coldwind showed that time-of-check-to-time-of-use race conditions on the userland kernel boundary can be found by analysing memory access patterns at runtime. The solution they created is called Bochspwn. It is an instrumentation module for the Bochs X86 emulator featuring an online and offline mode. Exploits for those double fetches differ in the raced value. It can either be a pointer or a buffer-size. Attacking buffer size is called an arithmetical race and more likely to succeed than attacking a pointer, which is more precise. The last section focused on widening the window of opportunity of an attack by having the raced value split across two pages. The winning idea is then, to only enable caches for the page the racing thread is modifying, while having the other thread fetch both pages.

By identifying, validating and getting new bugs closed, Bochspwn proved to be a viable solution for finding and analysing double fetch bugs within the real world.

References

[1] bochs: The Open Source IA-32 Emulation Project `http://bochs.sourceforge.net/`.

[2] Deeper into Windows Architecture `https://blogs.msdn.microsoft.com/hanybarakat/2007/02/25/deeper-into-windows-architecture/`.

[3] M. Jurczyk and G. Coldwind. Bochspwn - identifying 0-days via system-wide memory access pattern analysis. Black Hat USA `https://www.youtube.com/watch?v=ypV0kpi4cd8`, 2013.

[4] M. Jurczyk and G. Coldwind. Exploiting kernel race conditions found via memory access patterns. In *SyScan*, 2013.

[5] M. Jurczyk and G. Coldwind. Kernel double-fetch race condition exploitation on x86 further thoughts, jun 2013. `http://j00ru.vexillium.org/?p=1880`.

[6] M. Jurczyk, G. Coldwind, et al. Identifying and exploiting windows kernel race conditions via memory access patterns. 2013.

[7] F. J. Serna. Ms08-061 - the case of the kernel mode double-fetch, 2008. `https://blogs.technet.microsoft.com/srd/2008/10/14/ms08-061`.

[8] F. Wilhelm. Tracing privileged memory accesses to discover software vulnerabilities. 2015.