Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns

Mateusz Jurczyk, Gynvael Coldwind
Google Inc.

April 2013

Abstract

The overall security posture of operating systems' kernels – and specifically the Microsoft Windows NT kernel – against both local and remote attacks has visibly improved throughout the last decade. In our opinion, this is primarily due to the increasing interest in kernel-mode vulnerabilities by both white and black-hat parties, as they ultimately allow attackers to subvert the currently widespread defense-in-depth technologies implemented on operating system level, such as sandboxing, or other features enabling better management of privileges within the execution environment (e.g. *Mandatory Integrity Control*). As a direct outcome, Microsoft has invested considerable resources in both improving the development process with programs like *Secure Development Lifecycle*, and explicitly hardening the kernel against existing attacks; the latter was particularly characteristic to Windows 8, which introduced more kernel security improvements than any NT-family system thus far[11]. In this paper, we discuss the concept of employing CPU-level operating system instrumentation to identify potential instances of local race conditions in fetching user-mode input data within system call handlers and other user-facing ring-0 code, and how it was successfully implemented in the *Bochspwn* project. Further in the document, we present a number of generic techniques easing the exploitation of timing bound kernel vulnerabilities and show how these techniques can be employed in practical attacks against three exemplary vulnerabilities discovered by Bochspwn. In the last sections, we conclusively provide some suggestions on related research areas that haven't been fully explored and require further development.

1 Introduction

The Microsoft Windows NT kernel – as designed back in 1993 to run within the IA-32 Protected Mode – makes extensive use of privilege separation, by differentiating a restricted *user mode* used to execute plain application code and a fully privileged *kernel mode*, which maintains complete control over the machine[1] and is responsible for running the core of the operating system together with device drivers, which interact with physical devices, manage virtual and physical memory, process user-mode requests and perform other system-critical tasks. Furthermore, the NT kernel provides support for multi-threading, enabling multiple programs to share the available CPU resources in a manner that does not require applications to be aware of the undergoing thread scheduling. Last but not least, the kernel implements address space separation, splitting the overall available virtual address space into a user-mode accessible region, a kernel-mode accessible region and non-addressable space (x86-64 only). Most notably, the user-mode memory regions are separate for each process running in the system (creating an *illusion* of exclusive address space for each program) and can be accessed from both ring-3 and ring-0. On the other hand, the kernel-mode region of the address space is system wide (i.e. don't change between context switches)[2] and accessible only from ring-0. Typical address space layouts[3] used on 32-bit and 64-bit Windows platforms are illustrated in Figure 1.

In terms of the above, the user-mode portions of memory can be thought of as a shared resource. Indeed, these regions are accessible within both modes of execution, and due to context switching and multi-core hardware configurations, neither ring-3 nor ring-0 code can know for sure if another thread is modifying data at a specific user-mode address at any given point of time[4]. Simultaneous access to shared resources like memory is a well-known problem in the world of software development – by now, it has been widely discussed and resolved in numerous ways. In case of maintaining kernel state consistency while processing data originating from user-mode memory, there are two primary means to achieve a secure implementation: either ensure that each bit of the shared memory is fetched only once within each functional block of kernel code, or alternatively enforce access synchronization. As the latter option is difficult to implement correctly and comes at a significant performance cost, it is common to observe that when a kernel routine needs to operate on user-provided data,

[1] The Intel x86 architecture supports underlying modes of execution that have an even higher degree of control over the physical CPU, such as *Hypervisor Mode* or *System Management Mode*.

[2] One exception to the rule is the so–called "session memory", which may or may not be mapped within the kernel address space depending on whether the current thread is marked as GUI-enabled.

[3] On 32-bit versions of Windows, `/3GB` or `IncreaseUserVa` boot-time option can be used to enlarge the user-mode virtual address space to three gigabytes and limit the kernel-mode components to the remaining one gigabyte of memory.

[4] It is theoretically possible to ensure user-mode data integrity across a specific kernel code block by running at DISPATCH_LEVEL or higher IRQL on a single-core hardware configuration; however, this would require locking a specific memory area in physical memory, and would be plainly bad practice from Windows ring-0 development perspective.

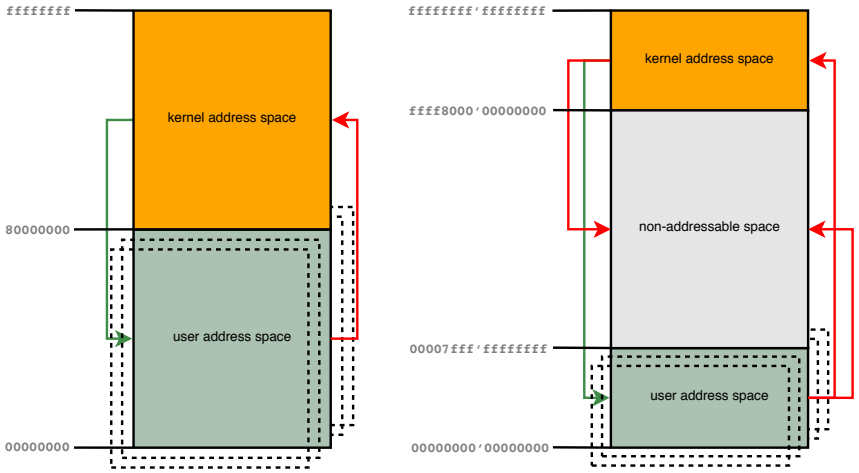

Figure 1: Microsoft Windows address space layouts in 32-bit and 64-bit modes.

it first copies the "shared" buffer into a kernel-mode allocation and uses the locally saved data for further processing. Since the kernel routine in question has exclusive ownership over the locally allocated buffer, data consistency and access races are no longer a concern.

As a direct outcome of the *client — server* architecture in user-mode applications interacting with the outside environment, moving data back and forth between the two types of memory regions happens all the time – ring-3 code only implements the program execution logic, while every interaction with files, registry, window manager or otherwise anything else managed by the operating system essentially boils down to *calling into* one of several hundred predefined kernel functions known as "system calls". Exhaustive lists of syscalls supported by each NT-family kernel since 1993 are available for reference[17][16].

Passing input parameters to system call handlers in both 32-bit and 64-bit versions of Windows is typically achieved as follows: the application pushes all necessary parameters on the user-mode stack similarly to invoking a regular routine using the *stdcall* or *cdecl* calling convention. After loading the service identifier into the `EAX` register, the code executes one of the `int 0x2e`, `sysenter` or `syscall` instructions, consequently switching the *Code Privilege Level* to 0 and transferring code execution into a generic syscall dispatcher (namely `nt!KiFastCallEntry` on modern Windows platforms). By using information stored in an internal "argument table", the dispatcher obtains the number of function parameters expected by the specific system call and copies $n * \{4, 8\}$ bytes (depending on the bitness) from the user-mode stack into the local kernel-mode one, consequently setting up a valid stack frame for the handler function, which is called next.

By doing the above, the generic syscall dispatcher prevents any memory-

sharing problems for the top–level information passed in by the user-mode caller – as each input parameter is referenced once while being captured and moved into the ring-0 stack, the syscall handlers themselves don't have to get back to user-mode to fetch their arguments or otherwise be concerned about the consistency of parameters over time. However, top-level parameters are very rarely used to represent actual input data – instead, they are commonly pointers to information (such as strings or other compound data types) stored in complex structures such as `UNICODE_STRING` or `SECURITY_DESCRIPTOR`. An example of passing an `OBJECT_ATTRIBUTES` structure to a system call handler is illustrated in Figure 2.

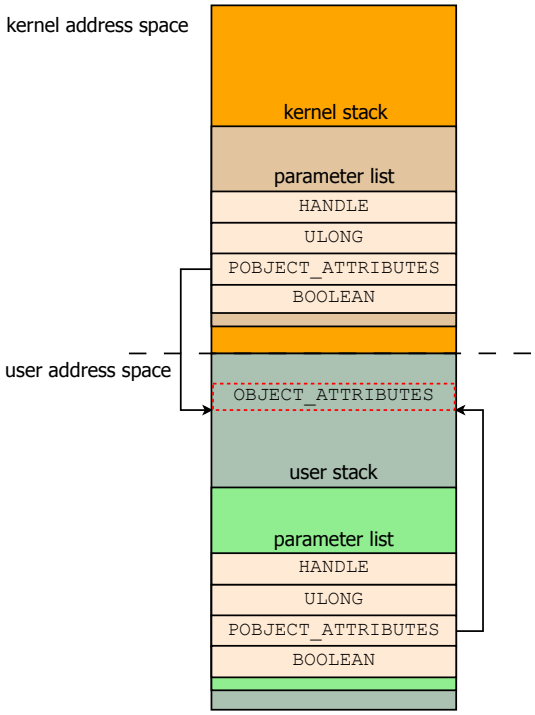

Figure 2: Example of system call parameter pointing back to user-mode.

The generic dispatcher is not aware of the specific types of data passed through parameters – therefore, all further fetches of memory pointed to by pointers in arguments are the responsibility of each system call handler. This opens up room for potential vulnerabilities – if any of the hundreds user-facing system calls or other code paths (e.g. IOCTL handlers in device drivers) lack proper transferring of input ring-3 memory regions to exclusively owned ring-0 buffers, then causing inconsistency in between subsequent memory reads can take a local attacker anywhere from a denial of service up to a local elevation

of privileges.

2 Race conditions in interactions with user-mode memory

Given what we already know about problems related to processing input data residing in the user-mode portions of virtual address space, let's explicitly state the erroneous condition we discuss in this paper. We are looking for consistent code paths (syscalls, IOCTL handlers etc.) referring to a specific user-mode address more than once in the same semantic context. For a single 32-bit input value referred to as x, a typical bug reveals itself in the following scenario:

1. Fetch x and establish an assumption based on its value.
2. Fetch x and use it in conformance with assumption from point 1.
3. Fetch x and use it in conformance with assumptions from points $\{1, 2\}$.
4. Fetch x and use it in conformance with assumptions from points $\{1, 2, 3\}$.
5. ...

We can classify each assumption made by a potentially vulnerable piece of code as belonging to one of two groups: directly and indirectly related to the x value. A direct assumption against the value of x is usually a result of sanity checking, i.e. verifying that the value meets specific criteria such as divisibility by two, being acceptably small or being a valid user-mode pointer. An example of directly enforcing specific characteristics over an input value is shown in Listing 1 in the form of a vulnerable Windows kernel-mode C code snippet.

Listing 1: Double fetch after sanitization of input value.

```
PDWORD *lpInputPtr = /* controlled user-mode address */;
UCHAR LocalBuffer[256];

if (*lpInputrPtr > sizeof(LocalBuffer)) {
  return STATUS_INVALID_PARAMETER;
}

RtlCopyMemory(LocalBuffer, lpInputPtr, *lpInputPtr);
```

Cases such as the one above can be thought of as *true* examples of *time of check to time of use* bugs, since both "check" and "use" stages are found in the actual implementation. Obviously, operating on data types other than integers (such as pointers) can also be subject to *time of check to time of use* issues – an instance of a `ProbeForWrite(*UserModePointer, ...)` call can be equally dangerous from a system security standpoint.

The other type of assumption frequently established while making use of user-supplied data can be observed when a value is taken as a factor for setting

up specific parts of the execution context, such as allocating a dynamically-sized buffer. In this case, no specific properties are directly enforced over the value in question; instead, certain relations between objects in memory are set up. Although no explicit sanitization takes place in the routine, breaking these implicit assumptions by changing the value of a supposedly consistent user-mode variable can nevertheless lead to serious issues such as exploitable memory corruption conditions. Listing 2 shows an example snippet of Windows device driver code which ties the size of a pool-based buffer to a value residing within user-mode, and later reuses that number as a parameter in a `RtlCopyMemory` call, effectively introducing a potential pool-based buffer overflow vulnerability.

Listing 2: Double fetch while allocating and filling out a kernel pool buffer.

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

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

Regardless of the exact type of assumption that user-land memory race conditions make it possible to violate, they can pose a significant threat to the security posture of a Windows-driven machine. While no such vulnerabilities are known to be exploitable remotely due to the fact that none of the remote clients have direct control over any of the local processes' virtual address space, we have observed that nearly every such issue can be taken advantage of locally, i.e. assuming that the attacker already has access to the machine through an existing account in the operating system.

In order to make the transition from a race condition to an information disclosure or memory corruption, further leveraged to execute arbitrary code, the attacker is required to "win the race", i.e. insert specially crafted data (often a 32-bit or 64-bit integer) into the memory region reused by the vulnerable kernel routine, and do it exactly in between the moment an assumption is established and usage of the data capable of breaking that assumption. For most real-life security flaws, this effectively boils down to targeting the time frame between the first and second ring-0 access to the memory in question, also referred to as the race condition *time window* further in this paper.

The size of the time window is typically measured in the total number of instructions executed between two subsequent accesses to the same memory area (a CPU-agnostic approach) or alternatively the time / cycles it takes to execute those separating instructions. Intuitively, the larger the window, the more probable it is to win the race in any particular attempt. As shown in the *Case Study* section, the time frames available to an attacker can vary from one to hundreds of instructions depending on the nature of the issue in question.

The fundamental problem in making practical use of race condition vulnerabilities is how strictly they are bound to minimal timings, the task scheduling algorithm implemented by the kernel and other countless characteristics of the execution environment (such as number of logged in users, running applications, CPU load, network traffic, ...) that all affect how the microprocessor's computation resources are shared across threads. Fortunately, failed attempts to win a race rarely cause any damage to the target software execution flow, thus allowing multiple attempts per second to be carried out before the goal is eventually accomplished.

A simplified version of the context switch order enabling an attacker to successfully exploit a double-fetch vulnerability on a single core hardware configuration is shown in Figure 3.

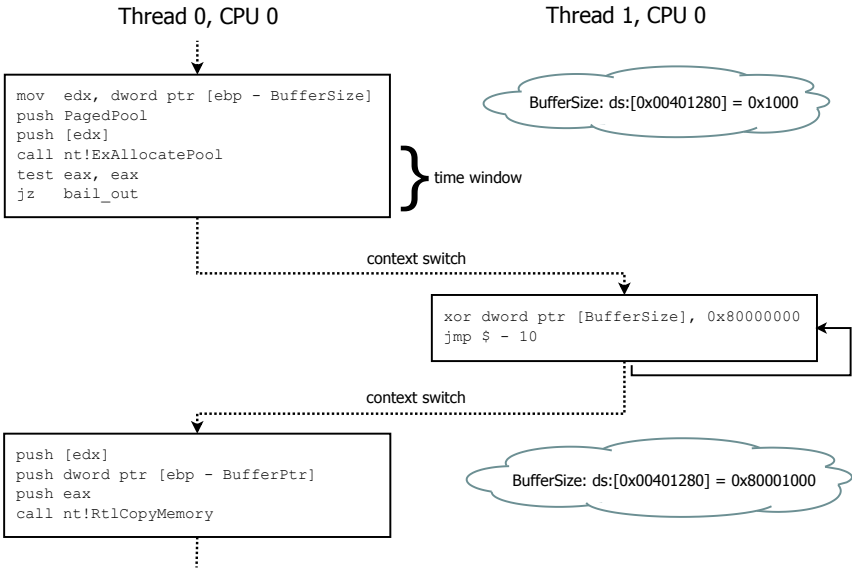

Figure 3: Context switch order allowing exploitation of a double-fetch vulnerability on a single CPU.

With just a single logical CPU, it is not possible to flip bits in the repeatedly fetched memory region *truly* in parallel with the affected kernel function. Therefore, it is essential that the same final effect is achieved by having the system task scheduler to preempt the thread residing in ring-0 within the time window between two subsequent fetches, and transfer code execution to attacker-controlled code. Considering that user-mode applications have rather limited means of controlling the scheduler's behavior (mostly consisting of tampering with priority classes and thread spraying), the size of the attack window plays a crucial role in assessing the exploitability of any such race condition. The *Exploitation* section discusses several techniques capable of improving the odds

of winning a race on a single CPU platform.

The situation is significantly more favorable to an attacker on hardware configurations including two or more logical CPUs (due to multiple physical CPUs, or more popular in modern hardware multiple cores and/or the Hyper-Threading technology), which seem to cover a great majority of both desktop and server platforms used at the time of this writing. In this scenario, an attacker can in fact utilize multiple cores to ensure that the memory being raced against is simultaneously processed by the vulnerable kernel routine and modified by other malicious threads with distinct core affinity masks. Reliability-wise, this change makes a huge difference, as the available time window no longer denotes a range of code paths within which execution control must be transferred to an attacker's thread, but instead describes the time frame during which a specific instruction (e.g. `XOR` or `MOV`) must be executed in parallel on a different processor. If the memory operation is plainly looped in the *flipping thread* (i.e. the thread responsible for continuously flipping bits in the targeted value), it becomes possible to achieve a reasonably high ratio of race wins per second even for the smallest time windows imaginable. In general, we believe that while certain race conditions in the kernel can be concerned as non-exploitable on single-core platforms, the availability of at least one more physical core makes any such issue fully exploitable. One example of practical exploitation of an extremely constrained vulnerability is addressed in the *Case study* section.

Figure 4 illustrates the optimal thread assignment and code flow granting successful exploitation using two logical CPUs to conduct an attack.

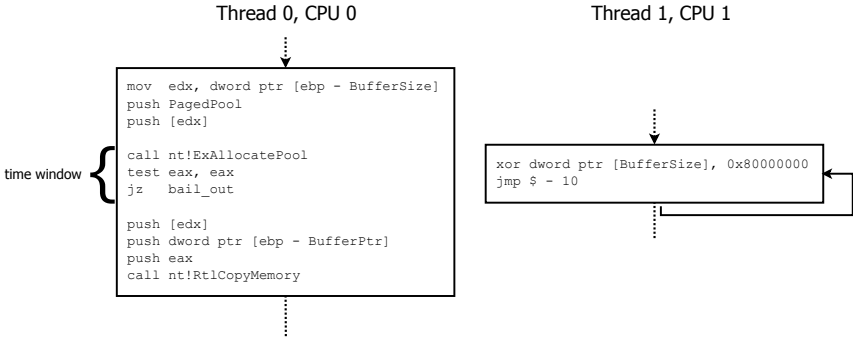

Figure 4: Example task scheduling allowing exploitation of a double-fetch vulnerability on two physical execution units.

While potentially more difficult to exploit when compared to the typical, direct memory corruption issues such as buffer overflows, race conditions in interacting with user-mode memory are also believed to be non-trivial to find. According to the authors' knowledge, there are currently no publicly available tools capable of detecting the specific class of problems in operating systems'

kernels. Processing certain types of data several levels deep into the NT kernel image frequently involves operating on both user and kernel-mode memory buffers at the same time, thus spotting the usage of a single ring-3 region in multiple places within a syscall might turn out harder than it appears (not to mention it is a dubious task). This might explain why – despite Microsoft being well aware of the presence of the vulnerability class in the Windows kernel – a great number of trivial race conditions have remained under the radar and were only fixed in 2013 after being detected by Bochspwn.

Double-fetch security flaws are relatively new and unique in the world of Microsoft; however, they otherwise have a rather long history and have been discussed multiple times in the context of other operating system' security. The next section outlines some of the work that has already been done in the past in the area of hardening user/kernel memory interactions.

2.1 Prior research

Until February 2013, the only publicly discussed instance of a *double fetch* vulnerability in the Windows kernel that we are aware of was CVE-2008-2252, a bug reported by Thomas Garnier back in 2008 and fixed by Microsoft as part of the MS08-061 security bulletin. The nature of the vulnerability was apparently unique enough to warrant a separate post on the *Microsoft Security Research & Defense* blog written and published by Fermin Serna on the Patch Tuesday itself[6]. Although the bug represented a fresh type of issue that could have been found in the kernel with relative ease, the incident remained a one-off event and didn't seem to have any visible consequences. In addition to the blog post, another short article regarding issues in capturing input data from kernel mode was published by Jonathan Morrison[10], a developer in the Microsoft's Core Operating Systems group in 2008, which makes it clear that the vendor was in fact internally aware of the class of problems.

Time of check to time of use (*toctou* in short) vulnerabilities appear to be better audited for in the Linux kernel[5] – one of the most recent and public instances of the discussed type of bug was reported by Alexander Viro in 2005[5]. The flaw was found in the implementation of a `sendmsg` syscall and effectively allowed for a stack-based buffer overflow to occur, thus having significant security implications. An in-depth write up covering the exploitation process was provided by *sgrakkyu* and *twiz* in the *Attacking the Core: Kernel Exploiting Notes* Phrack article in 2007[31].

Further, fundamentally similar attack vectors exist in the Linux kernel due to the fact that it supports an otherwise uncommon user-mode ability to trace and control system call invocations for other processes in the system. The feature is widely used for numerous security-related purposes, such as monitoring the performance of known malicious applications, sandboxing renderer processes for

[5]Note that it is not clear whether Linux is in fact more robust than other operating system kernels when confronted with Bochspwn, as the tool has not been used against any kernel other than Windows itself. The results of further testing involving different software platforms are going to be published shortly.

common attack targets or emulating a chroot environment[6]. While it can be used to effectively improve the security posture of the execution environment, it also comes with potential risks – if not implemented carefully, the ptraceing process could allow for inconsistency of the syscall parameters in between the time of custom sanitization and the time of having them processed by the kernel itself. The problem was discussed as early as in 2003 by Tal Garfinkel[32] (Section 4.3.3, "system call races"), Niels Provos in 2004[28], Robert N. M. Watson in 2007[30] and many more.

We are currently not aware of any relevant research performed in this area for BSD-based or any other operating systems.

3 The Bochspwn project

Historically, the primary Windows user/gdi device driver (`win32k.sys`) has been one of the most prominent sources of local security issues identified in the Windows kernel, with a volume of vulnerabilities strongly exceeding those in the kernel image or any other default driver in the operating system. As a result, the implementation of various potentially sensitive features found in the module are subject to regular assembly-level code reviews by security professionals around the world. While going through the implementation of an internal `win32k!SfnINOUTSTYLECHANGE` routine in 2008, we have identified what appeared to be a typical example of a double fetch bug[7]. While further investigation of the finding was dropped at the time, the very same issue was rediscovered in October 2012 and examined in more detail. Once exploitability of the bug was confirmed, we further determined that the very same pattern was shared across 26 other internal `win32k.sys` functions. As much as the existence and volume of the issues surprised us, we decided to start a large effort focused on identifying other potential double-fetch vulnerabilities across the overall codebase of Windows ring-0. The following subsections discuss why we eventually decided to pursue a dynamic approach, and how it was implemented using the instrumentation API provided by the *Bochs* x86-64 CPU software emulator.

3.1 Memory Access Pattern Analysis

While manual code audits allows one to successfully discover some bugs of the double-fetch class, it is both time consuming and does not scale well. Hence, in order to effectively identify such vulnerabilities, we decided to develop a set of tools that would allow us to cover vast areas of kernel-mode code while maintaining a low false positive ratio.

As explained earlier, the double-fetch vulnerability class manifests itself with two distinct same-size memory reads from the exact same user-mode memory

[6] See the PRoot project – http://proot.me/

[7] A case study, including a complete exploitation process of the vulnerability is found in Section 5.1, *CVE-2013-1254*

location, with both *read* events taking place within a short time frame. The pattern can be detected using both static and dynamic (runtime) analysis of the operating systems kernel and kernel-mode drivers. While the first solution offers excessive coverage including rarely used areas of code, we have chosen to follow the dynamic analysis model due to the shorter tool development period and generally simpler design, as well as the existence of open source software which could be easily modified to facilitate our needs.

Looking at the problem from a dynamic analysis perspective, one crucial requirement enabling one to discover the said pattern is the ability to monitor and log memory reads performed by an operating system.[8] – the task can be achieved in several different ways, as further discussed in the *Design* section.

3.2 Double-fetch pattern

Defining the double-fetch pattern as simply two consequent reads of the same memory location is not in fact sufficient to achieve satisfying results; instead, it is required to define the pattern in greater detail. The general conditions a memory access must meet in order to potentially manifest or otherwise be a part of a double-fetch vulnerability are as follows:

- There are at least two memory reads from the same virtual address.
- Both read operations take place within a short time frame (not necessarily expressed in time units).
- The code instantiating the reads must execute in kernel mode.
- The virtual address subject to multiple reads must reside in memory writable by ring-3 threads (i.e. user-mode portions of the address space).

The following subsections carefully describe each of the points listed above, including Windows-specific implementations of the constraints' verification and their overall influence on the volume of false positives they might yield.

3.2.1 Multiple memory reads from the same address

The essence of this point is rather self explanatory; however, it still leaves the relations between the sizes of subsequent memory accesses undetermined. As previously mentioned, it is important that both values fetched during the multiple reads are used in the same semantic context, i.e. are assumed to be equal across the memory references. In typical C code, the four most commonly observed reasons for accessing memory are:

1. Referencing a single variable or a pointer to one.
2. Referencing a structure field.

[8] Technically, monitoring references to virtual memory is the desired approach, as only monitoring physical memory access would increase the complexity of the design.

3. Referencing an array.

4. A combination of the above, e.g. accessing a structure field within an array.

In all of the above cases, the code operates on an abstract object of simple type – an integer (signed or unsigned, normally between one to eight bytes long) or a floating point number (normally between four to ten bytes). If the high level C code references the same object twice, it must have the same type in both cases, thus both reads are always of the same size. On the other hand, experiencing consecutive, differently-sized reads from the same location may indicate any of the following:

1. The code attempts to reference union fields, which are aligned at the same offset, but have different sizes. This is an extremely uncommon scenario while interacting with user-mode memory.

2. The code invokes a function operating on memory blobs, such as `memcmp` or `memcpy`. In such case, the size of each particular read used to move or compare the bitstreams are implementation-specific and can be different from the sizes of particular fields of a structure or array subject to the memory operation.

3. The first read from a memory location is purposed to verify that the virtual address is backed by physical memory, e.g. a special, inlined version of a `ProbeForWrite` function call, such as the one presented in Listing 3.

Listing 3: Input address sanitization found in NtAllocateVirtualMemory

```
PAGE:0061A838                   mov     ecx, [ebp+AllocationSize]
PAGE:0061A83B                   mov     eax, ds:_MmUserProbeAddress
PAGE:0061A840                   cmp     ecx, eax
PAGE:0061A842                   jb      short loc_61A846
PAGE:0061A844                   mov     ecx, eax
PAGE:0061A846
PAGE:0061A846 loc_61A846:
PAGE:0061A846                   mov     eax, [ecx]
PAGE:0061A848                   mov     [ecx], eax
```

In case of the first item, accessing different union fields which happen to reside at the same location does not comply with the "same semantic context" rule, and therefore is not interesting. Similarly, referencing a memory area for the sole purpose of verifying that it exists rather than fetching the underlying value cannot be part of a double-fetch vulnerability. Consequently, two respective reads from the same location are only considered a potential issue if both have equal length, or at least one of them originates from a standard C library memory handling function.

Even still, following the above rules does not guarantee exclusively true positives – please consider the example in Listing 4. As the code snippet illustrates,

it is required that the function not only fetches a value twice, but also makes use of at least one common bit shared between the two numbers, which is not the case in the example.

Listing 4: A same-sized false positive.

```
mov eax, [esi]
and eax, 0xff00ff00
...
mov eax, [esi]
and eax, 0x00ff00ff
...
```

In general, it is extremely uncommon for the kernel to fetch data from user-mode in a single byte granularity – a great majority of array or structure fields are at least two bytes in length. Given that many variations of the probing functions include one-byte accesses, which won't ever result in the discovery of a double fetch but still increase the CPU and memory complexity of the instrumentation process, we decided to disregard reads of size=1 entirely, thus optimizing the project run time without sacrificing information regarding actual vulnerabilities.

Furthermore, it is important to note the distinction between the *read-only* and *read-modify-write* instructions. While the first group only reads memory at a specific memory address and copies it into a specific place in the CPU context (e.g. a register) so that it can be further operated upon, the latter ones don't actually reveal the fetched value *outside* of the instruction, but only use it within the scope of the instruction. Examples of both groups can be `mov eax, [ecx]` and `and [eax], 0`, respectively.

As a consequence of the fact that it is required for the fetched value to be further used in the vulnerable assembly code beyond just the fetching instruction, none of the *read-modify-write* family commands can ever be a part of a double fetch. One notable exception to the rule are instructions which while don't explicitly load the obtained number anywhere, still give some indications regarding the value through specific CPU flags, such as the *Zero Flag* or *Parity Flag*. For instance, the C code snippet shown in Listing 5 can in fact be compiled to a `DEC` instruction, which (although it belongs to the *read-modify-write* group) could indeed be part of a double fetch, due to information about the decremented number being zero is *leaked* through ZF and immediately used in the subsequent conditional branch, as presented in Listing 6. We have determined that both gcc (version 4.7.2) with the `-O3` flag and clang (version 3.2) with the `-O3` or `-Os` flags are capable of performing such optimizations; we believe the "Microsoft C/C++ Optimizing Compiler" could also generate similar assembly code.

Listing 5: Code modifying a variable and testing its value at the same time.

```
if (!--(*ptr)) {
  puts("Hello, world!");
}
```

Listing 6: Using a read-modify-write instruction to test a memory value.

```
8048477:       ff 4c 24 08             decl   0x8(%esp)
804847b:       75 0c                   jne    8048489 <main+0x29>
```

3.2.2 A short time frame

It is somewhat intuitive what the "short time frame" term is supposed to mean – if both reads are expected to have the same semantics in the context of the vulnerable code, they must both be triggered within a shared call tree. For the sake of an accurate implementation, we need to further specify the frame of reference and how the "short" part is actually measured. As it turns out, using actual time or instruction counts as indicators of whether two memory reads are related might not be the best idea, given that modern multi-tasking operating systems continuously switch between different execution units (threads), which often results in changing the virtual address space to that of another process.

Considering what we already know about double fetches, it is apparent that it only makes sense to limit the frame of reference to a single thread, e.g. examine sequences of kernel-mode memory reads on a per-thread basis. In the Windows operating system[9], it is not sufficient to represent unique threads by only using a pair of process and thread IDs, as both identifiers are extensively reused by the operating system for different objects during normal execution, once their previous occupants are terminated or exit gracefully. In order to ensure that each separate thread living in the system is represented by a unique tuple, we also use the thread creation time as the third value of the internal thread identifier. No two threads are ever created with the same PID and TID at the very same time.

Once the reference frame has been established, we can continue to further specify the *short time frame* term. In our research, we initially assumed that the distance between reads from the same address would be measured in the number of distinct reads from kernel-mode code to user-mode memory. Using this measurement method, a double-fetch vulnerability was be defined as two reads separated by "at most N distinct reads to other addresses", with N ranging from 5 to 20. This solution, while clearly having certain cons, was able to successfully identify several vulnerabilities in Windows kernel and `win32k` subsystem module. However, there are several important downsides of this method:

1. A real double-fetch bug might go undetected if the two reads are separated with $N+1$ or more other reads. The volume of undetected "real" positives increases with smaller values of N.

2. A false positive might be yielded if both reads belong to two separate consequent system calls, i.e. a return from one system call takes places, and another system call is invoked, operating on the same or similar set of parameters – both handlers would access the same user-mode memory

[9]The statement likely holds true for other common platforms, too.

area, but it would not be a double-fetch vulnerability due to different semantic contexts. The volume of false positives increases proportionally to the value of N.

To address the problem, we have conceived a different approach based on defining the problem as "two reads taking place during the handling of a single system call". This method requires keeping a larger, variable sized cache of memory reads, as well as implementing additional instruction-level instrumentation, in order to detect occurrences of the `sysenter` and `sysexit` (or equivalent) instructions.

There are very few cases in which this method could throw false positives. One theoretical example would be the event of a hardware interrupt preempting the currently running thread without performing a context switch and attempting to read user-mode memory; this, however, makes little sense for an operating system and is very unlikely to be observed in any real platforms.

While the approach only results in a manageable increase in the complexity of the project, it has proven to be by far the most effective one throughout several iterations of testing – as such, we successfully used it during four out of five iterations carried out against Windows 7 and 8.

3.2.3 Kernel-mode code

In our research, we have defined the memory referencing code as any code running with *ring-0* privileges (i.e. at CPL=0) – including the kernel image itself (e.g. `ntoskrnl.exe` or `ntkrnlpa.exe`), as well as all other kernel modules (e.g. `win32k.sys` or third-party device drivers) and code executed outside of any executable module (e.g. the boot loader).

The above definition is trivial to implement and cheap in execution time. On the other hand, it is prone to certain classes of false positives:

- Vulnerable kernel-mode code might not necessarily be invokable (neither directly or indirectly) by a process owned by a non-privileged user – this makes the existence of such bugs uninteresting in the context of identifying classic *elevation of privilege* vulnerabilities. However, in operating systems that enforce signing of device drivers (e.g. Windows 7 64-bit in default run mode), such issues can be still potentially used to carry out *admin to ring-0* attacks, and as such should not be considered a false positive.

- During the booting and system initialization phases, user-mode portions of certain core processes (such as smss.exe in Windows) can be accessed by kernel code for legitimate reasons. However, since it is not possible for user-mode code to trigger the code areas at post-boot time, they are beyond our interest.

Additionally, by narrowing the execution origin to ring-0, the following real positives cannot be detected:

- System or administrator-owned processes interacting with an untrusted process via shared memory sections.
- Hypervisor mode and SMM double fetches.

Note that both variants are very case-specific and extremely difficult to accommodate in Bochspwn without introducing fundamental changes to the project's design.

3.2.4 User-controlled memory

The "user-controlled" memory portions are defined as part of the virtual space which can be written to by ring-3 code. In case of Windows, this covers all addresses from 0 up to the `KUSER_SHARED_DATA` region, which is the highest-mapped page in the userland address space, and is not writeable in itself.

The most elegant way to verify the condition would be to look up the page table entry for the virtual address in question, and use it to check if it is marked as user-accessible. The problem with this solution is that it comes at a performance cost – identifying the address descriptor structure in Bochs process memory and testing it is an expensive operation. To address the problem, we chose to make use of an extremely fast method specific to the memory layout found in Microsoft Windows platforms; namely, test if the referenced address is within user-mode virtual address space boundaries. In practice, this was accomplished by testing the most significant bit of the address on 32-bit systems, and 16 most significant bits on 64-bit platforms.

One might argue that certain areas of kernel memory can be indirectly influenced by user-mode code and as such could also create a base for double-fetch problems. However, in such case, the vulnerability would be caused by the lack or improper synchronization of access to kernel-mode structures, which is a separate vulnerability class completely and not discussed in this paper.

3.3 Design

We have considered several possible approaches to create a system-wide memory monitoring tool, with each potential solution differing in performance and invasiveness, as well as features offered "out of the box" and simplicity of both design and implementation:

- Modifying the system exception handler to intercept memory references by using either segment or page access rights. The method would require introducing significant, internal modifications in the kernel of the operating system in question, which unfortunately makes it the least portable option, especially difficult to implement for the Windows platform. At the same time, it would have a minimal CPU overhead and would likely allow to monitor an operating system on physical hardware.
- Making use of a thin hypervisor with similar means of intercepting memory references as the previous idea. This method is less invasive in that it

does not require hacking kernels on a per-system basis. However, the development and testing process of a functional hypervisor with extensive logging capabilities is a complex and time consuming task; therefore, it was considered unsuitable for a prototype we wanted to create.

- Employing a full-blown CPU emulator to run an operating system, and instrument the memory references at the software level. While this was certainly the worst solution performance-wise and also had several other shortcomings (e.g. inability to test certain real hardware-specific kernel modules such as modern video card drivers), it is the least invasive, most elegant and simplest to quickly implement and test.

After considering the pros and cons of all options, we initially decided to follow the full CPU emulator approach due to its overwhelming advantages. Bochspwn, our system-wide memory monitoring tool, is based on the open-source Bochs x86 emulator and takes advantage of Bochs' instrumentation[10] API. Even though the original goal of the project was to specifically target double-fetch patterns, the overall design is flexible and not limited to this specific class of software vulnerabilities.

Bochspwn operates in one of two modes:

- **offline** – all user-mode memory accesses originating from kernel mode are saved to external storage at emulation run time, with the actual scanning for double-fetch issues performed post-instrumentation. Since the instrumentation does not include the access pattern logic, it is more effective in terms of CPU time used (and therefore allows the guest system to execute with higher *Instructions Per Second* rates). On the other hand, the mode heavily charges storage space, reaching tens or hundreds of gigabytes for a single log file (see the *Performance* section for details).

- **online** – the double-fetch checks are performed at instrumentation run time. As the double fetch identification algorithm is applied to memory operations as they occur, only information about vulnerability candidates is saved to disk, thus saving a lot of storage space. This is achieved at the cost of increased CPU and memory usage, as it is required to keep per-thread caches in memory and process them as needed.neglectable

While the designs of both modes share common parts, they are generally implemented distinctively (e.g. offline-mode requires additional external tools for log analysis). The following subsections discuss the different parts of Bochspwn in more detail.

[10]The Bochs instrumentation feature is essentially a set of preprocessor macros, invoked by Bochs during the emulation of different CPU events, such as fetching a new instruction from memory or performing linear memory access. By attaching actual functions to some of the macros, the instrumentation grants itself the ability to monitor certain portions of the CPU functionality.

3.3.1 Common parts

Both modes share the same instrumentation interface, as well as code responsible for gathering actual data.

Bochspwn instruments the two following Bochs events[11]:

- `BX_INSTR_BEFORE_EXECUTION` – invoked prior to emulating the next CPU instruction.
- `BX_INSTR_LIN_ACCESS` – invoked for each linear memory access taking place in the virtual CPU.

The first event is used exclusively to detect instances of the `syscall` and `sysenter` instructions being executed. This is required to implement the "short time frame" constrain of the double-fetch memory pattern – we use this event to differentiate between distinct system call handlers. The event is either logged to a file in case of the offline mode, or triggers a memory-access read cache flush in the online mode.

The second instrumentation macro is the most important one for the overall memory access pattern analysis, as it is the primary source of information regarding all memory operations taking place in the emulated environment. The callback is provided detailed data about the memory access, such as the linear address, physical address, length of operation and type of access (one of `BX_READ`, `BX_WRITE`, `BX_RW`[12]). The event handler starts off by performing initial checks whose sole purpose is to discard the non-interesting memory reads as soon as possible, prior to examining the operation in terms of a potential double fetch situation. These checks implement some of the previously discussed double fetch pattern constraints, and were selected based on their simplicity. The criteria taken into account during initial verification are as follows:

- Check if the CPU is in 32 or 64-bit protected mode.
- Check if the access type is `BX_READ`.
- Check if the access originates from kernel-mode code.
- Check if the access size is within desired limits (e.g. two to eight bytes).
- Check if the address resides within user-mode.

The remaining constrains – "same memory addresses" and "a short time frame" – are tested in either a separate function in online mode, or by external tools in case of the offline mode.

Furthermore, the common code is responsible for acquiring additional system-specific information about the source of the linear memory access event. This information is acquired by traversing the guest operating system memory and reading certain kernel structures containing the desired data, such as:

[11] Also `BX_INSTR_INITIALIZE` and `BX_INSTR_EXIT` are used for initialization and deinitialization purposes, though this it not relevant to memory analysis.

[12] The `BX_RW` access type is set in case of *read-modify-write* instructions (e.g. `inc [ebx]`). Certain faulty versions of Bochs incorrectly report the `BX_READ` access type instead of `BX_RW`.

- Process image file name, e.g. `explorer.exe`.
- Process ID.
- Thread ID.
- Thread creation time.
- Execution call stack information, including:
  - Name of kernel-mode module, e.g. `win32k.sys`.
  - In-memory image base address of the module.
  - Instruction pointer expressed as a relative offset inside the module.

The purpose of acquiring the above types of information is twofold. Firstly, the process ID, thread ID and thread creation time form a tuple used as a unique thread identifier for the purpose of separating memory reads across different execution units. Secondly, detailed data regarding the context of the read (such as an accurate stack trace) prove extremely useful during the vulnerability categorization and deduplication stages, as well as for establishing the actual root cause of an issue. For example, imagine that the second read in a double fetch condition takes place inside of the `memcpy` function call – without information about the caller's location, it is often impossible to determine where the routine was invoked, and thus what the vulnerability is.

An important thing to note here is that acquiring the above information is obviously a costly process. Each read from guest memory requires, similarly to a physical chip, translating linear addresses to physical memory addresses, and then fetching the data from the Bochs' memory module internal buffers. In an effort to reduce the incurred overhead to a minimum, we implemented a configurable limit over the maximum number of stack trace entries acquired for each interesting memory read, and used values between 4 and 6 throughout all testing iterations. Unfortunately, 64-bit versions of Windows no longer maintain a stack layout allowing for the reconstruction of the complete call chain; in order to achieve the effect, additional debugging information is required for the executable module in consideration. As a result, we decided to only log information about the current frame (reducing the amount of data available while investigating detected problems) for 64-bit Windows.

3.3.2 Offline mode

Bochspwn in offline mode records all events meeting the initial criteria to a Protocol Buffer[13] log file, with no additional processing of the gathered data. This allows one to *record* an entire system session for a single time, and later experiment with different log processing tools and memory access patterns, without the need to repeat the time consuming data collection process. Additionally,

[13] https://code.google.com/p/protobuf/

in contrast to the online mode, there is a smaller CPU and memory footprint during emulation, hence the initial stage is faster.

The final log, essentially being a system-wide stream of memory events, requires further post-processing before it can yield meaningful results. For instance, we have been extensively using the following filters over the output logs prior to running the final double-fetch filter.

- Unique thread separation – separating the single input log into several (typically 1000 to 4000) files containing events assigned to particular threads.
- Removing noise – it turns out that certain Windows kernel device drivers tend to trigger a large number of false positives and no actual bugs. In such case, it might make sense to completely filter out entries related to such modules early on, so that they are not a concern in further phases of the process. One example of a *noisy* driver we encountered during testing was the `CI.dll` executable image, which we decided to exclude from our logs completely.
- Symbolization – presenting the output logs in a human-readable form, found to be especially useful for debugging purposes.

Once the original log file is cleaned up and split into many smaller streams of memory access information, those files become subject to the final filter (in the form of a dedicated application), which examines all user-mode memory fetches performed within each respective system call invoked by the examined thread, in search for instances of the same address being used twice (using the same length and meeting the other mandatory conditions, as described in the previous sections). Once the processing of all files is complete, the output results require further manual investigation, as a number of double fetch candidates can still turn out to be false positives or duplicates of already known issues. An exemplary output entry yielded during the first testing iteration, which exhibited an actual vulnerability in the `win32k!SfnINOUTNCCALCSIZE` routine fixed by Microsoft in February 2013 is shown below:

```
------------------------------ found double-read of address 0x000000000581f13c
Read no. 1:
[pid/tid/ct: 00000478/00000684/01ce00c754b25570] {    explorer.exe}
             00000057, 000011fc: READ of 581f13c (1 * 4 bytes),
             pc = 8fb21d42 [ mov edx, dword ptr ds:[edx] ]
#0  0x8fb21d42 ((00116d42) win32k!SfnINOUTNCCALCSIZE+0000021e)
#1  0x8fb27e6e ((0011ce6e) win32k!xxxDefWindowProc+0000009a)
#2  0x8fb4365b ((0013865b) win32k!xxxSendMessageTimeout+00000329)
#3  0x8fb43912 ((00138912) win32k!xxxSendMessage+0000002c)
#4  0x8fb1fa84 ((00114a84) win32k!xxxCreateWindowEx+00000d93)
#5  0x8fb2274f ((0011774f) win32k!NtUserCreateWindowEx+0000032a)

Read no. 2:
[pid/tid/ct: 00000478/00000684/01ce00c754b25570] {    explorer.exe}
             00000057, 000011fc: READ of 581f13c (1 * 4 bytes),
             pc = 8fb21d9c [ mov eax, dword ptr ds:[ebx] ]
#0  0x8fb21d9c ((00116d9c) win32k!SfnINOUTNCCALCSIZE+00000278)
#1  0x8fb27e6e ((0011ce6e) win32k!xxxDefWindowProc+0000009a)
#2  0x8fb4365b ((0013865b) win32k!xxxSendMessageTimeout+00000329)
#3  0x8fb43912 ((00138912) win32k!xxxSendMessage+0000002c)
#4  0x8fb1fa84 ((00114a84) win32k!xxxCreateWindowEx+00000d93)
#5  0x8fb2274f ((0011774f) win32k!NtUserCreateWindowEx+0000032a)
```

3.3.3 Online mode

The online mode implements memory pattern scanning at guest OS run time, providing real-time results, saving disk space and reducing the volume of host I/O operations. At the same time, it requires a larger amount of memory in order to store the per-thread cache of recent memory accesses, and takes significantly more time to complete.

When a new system call invocation is detected through instrumentation of the `sysenter` instruction, the list of pending memory read entries associated with the current thread, which occurred in the previous syscall handler, is carefully examined for the presence of double fetch candidates – if any are found, the information is saved to external storage. Once the examination is complete, the current cache for that thread is flushed.

While online mode is significantly slower than the offline mode during data gathering phase, it does not require a post-processing phase. However, due to the fact that information about the reads is *lost* during the instrumentation process, any additional filtering has to be implemented in Bochspwn itself, which further pushes the CPU and/or memory overhead.

Overall, the online mode is suitable for long-running tests which would otherwise consume extensive amounts of disk space, as well as for continuous testing, e.g. using a driver or syscall fuzzer to increase the kernel code coverage.

3.4 Performance

In this section, we discuss the general performance of the project based on results obtained during several iterations of instrumentation of the Windows operating system, including memory and disk space consumption in different modes of execution. The aim of this section is to provide a general overview of CPU-level instrumentation efficiency; many of the numbers presented here are approximate, and therefore should not be considered as accurate benchmarks.

3.4.1 CPU overhead

In comparison to an operating system running on physical hardware, running Windows within a full-blown x86-64 software emulator which is additionally instrumented is extremely slow; especially so because of the fact that both `BX_INSTR_BEFORE_EXECUTION` and `BX_INSTR_LIN_ACCESS` intercepted events are one of the most fundamental and frequently invoked ones.

Table 3.1 shows the difference in boot time of a Windows 7 32-bit Starter Edition guest, measured from a cold boot to a fully loaded desktop on a Core i7-2600 @ 3.40 GHz host. In case of the offline mode, this only covers the log gathering step and none of the post-processing phases, nor double-fetch analysis.

Furthermore, Table 3.2 illustrates the volumes of different instrumented events, compared to how many of them pass the initial checks. The test run used to gather the information consisted of starting the operating system until the desktop was fully loaded, manually issuing a shutdown command inside of the guest, and ending the event counting once the guest reached power-off state.

Table 3.1: Time from a cold boot to a fully loaded desktop.

Description	**Time (mm:ss)**
Vanilla Bochs	04:25
Bochspwn offline mode	19:21
Bochspwn online mode	35:48

Table 3.2: Number of events from cold boot to system shutdown.

Description	**Number of events**
Instructions executed	20 160 778 164
`sysenter` or `syscall` instructions executed	804 084
Linear memory accesses	10 018 563 655
Linear memory accesses meeting initial criteria	32 852 185

The two following subsections cover the nature of differences observed between the online and offline mode overheads.

3.4.2 Offline mode overhead

As mentioned in the *Design* section, enabling the offline mode causes all events which pass the initial criteria to be recorded to a log file, with each entry consuming approximately from 150 to 250 bytes. Due to a large amount of reads passing the initial validation – reaching hundreds of millions of events on a well tested system – the load of I/O operations and disk space usage are far from being neglectable. Table 3.3 shows exemplary real-world sizes of the final log files obtained in the process of instrumenting Windows 7 and 8.

With regards to I/O operations, Bochspwn uses the standard buffering options offered by the C runtime library and the operating system to minimize

Table 3.3: Event log disk space usage experienced during our experiments.

System	Size of event logs	Note
Windows 7 32-bit	13 GB	Initial Bochspwn tests.
Windows 7 32-bit	78 GB	Full guest test suite.
Windows 8 64-bit	136 GB	Full guest test suite.
Windows 8 32-bit	107 GB	Full guest test suite.

the performance loss due to disk writes. This could be further extended to use asynchronous operations, or increase the buffering beyond standard limits.

As previously mentioned, the post-processing is specific to offline mode and consists of a set of steps during which additional filtering takes place. Each of the filtering steps needs to read the complete log from the disk at least once[14], process the input data and store the results back on the disk. During our research, we used different filters whose completion time immensely depended on the size of the log file(s) and could reach up to several minutes. Table 3.4 provides additional information on the particular filters' processing complexity and size of output data.

Table 3.4: Performance note on exemplary filters.

Filter	Processing	Output size
Unique-thread separation	Linear	Same as input log.
Removing certain events	Linear	Smaller than the input log.
Double-fetch scan	Linearithmic	Minimal.

To summarize, the offline mode trades off disk space and volume of disk accesses in exchange for reduced CPU and memory overhead during the monitoring phase.

3.4.3 Online mode overhead

In contrast to offline mode, the online one uses external storage only to save the final results (i.e. double fetch candidates), so the required disk space is minimal, similarly to the amount of time spent on I/O operations. However, this requires the steps that are normally found in the post-processing code to be done real-time. This incurs additional overhead, both in terms of memory (see Table 3.5) and CPU usage.

As shown in Table 3.1, Bochspwn online mode is roughly two times slower than offline mode (including the I/O communications performed in offline), yet it provides the results in real time and does not require neither excessive disk space nor additional log processing steps.

[14]In case of smaller test logs, one can use either a RAM-disk or rely on in-memory file caching offered by modern systems for faster read access.

Table 3.5: Memory usage peak for guest with 1024 MB of RAM.

Description	Memory usage peak
Vanilla Bochs	1 085 MB
Bochspwn offline mode	1 086 MB
Bochspwn online mode	1 584 MB

3.5 Testing and results

Initially, only offline mode was implemented in Bochspwn and used in the majority of our research. The early tests consisted of starting the operating system, running several applications available out of the box in Windows and shutting the system down. Those steps would result in having a 10-15 GB log file generated, which was then enough to test if detection of the double fetch pattern works in practice. As we already knew about the 27 previously discovered vulnerabilities in `win32k.sys`, we were able to reliably determine if the implemented pattern was indeed correct.

Having established a first working implementation, we began to improve the kernel code coverage by running various additional applications within the guest (e.g. programs making use of the DirectX API), as well as the Wine API unit-test suite[34] – as a result, this allowed us to cover additional system calls, device driver IOCTLs and various other executable regions in the kernel address space.

Furthermore, we added support for all Windows NT-family operating systems, including the 32 and 64-bit versions of Windows 8 and 64-bit version of Windows 7.

So far, we have conducted five complete iterations of testing, and accordingly reported the results of each phase to Microsoft (see table 3.6).

Table 3.6: A history of double fetch reports delivered to Microsoft.

Date	Iteration	Number of cases	Note
2012-09-19	-	27	Original issues discovered manually.
2012-11-01	1	20	Windows 7 32-bit.
2012-12-09	2	7	Windows 7 32-bit.
2012-12-16	3	20	Windows 8 32-bit.
2013-01-21	4	20	Windows 8 64-bit.
2013-01-30	5	22	Windows 8 32-bit.

At the time of this writing, the following numbers of cases have been resolved by the Redmond giant:

- In February 2013, Microsoft Security Bulletins MS13-016[19] and MS13-017[20] addressed 32 Elevation of Privilege issues, with three more fixed as variants.

- In April 2013, Microsoft Security Bulletins MS13-031[21] and MS13-036[22] addressed four Elevation of Privilege issues.
- 13 issues were classified as Local DoS only.
- 7 more issues are either scheduled to be fixed or are still being analyzed.
- The remaining reports were deemed to be either not exploitable, duplicates or were otherwise non-issues.

It is probable that further problems will be identified in future testing iterations, provided that a more efficient instrumentation will be eventually developed, or the code kernel coverage will be significantly improved.

4 Exploitation

The Bochspwn project provides effective means of identifying potential problems in sharing input data between user and kernel-mode. Further manual investigation helps to sieve the overall output report and find actual security flaws; yet, even with precise information of where the bugs reside and how to possibly trigger them, they are completely useless without a practical way to exploit them. This section outlines the various techniques that we have discovered or learned while reverse engineering and trying to take advantage of some of our project's findings – techniques that can be used to effectively compromise the security of a Microsoft Windows platform. While some of the methods described here are specific to races involving accessing ring-3 portions of the virtual address space, others can prove useful for other types of timing and/or scheduling-related problems.

It is important to note that winning a race condition, if one is found in an operating system, doesn't automatically grant you any superpowers in the executive environment without further work. Instead, kernel races are best thought of as *gateways* to actual violations which can be used in the attacker's favor, and winning a race only grants you the opportunity to exploit the specific type of violation, depending on what kernel code assumption is broken during the process. In that context, race conditions appear to be more complex in exploitation than the more "deterministic" classes of issues, as they require the attacker to prepare a strategy for both the race itself and post-race exploitation of the resulting condition.

While the details of each violation introduced by a double-fetch bug can differ from case to case depending on how the kernel uses the value raced against and what assumptions it establishes, there are four outcomes that are observed most frequently:

1. Buffer overflow – a result of inconsistency between the value used during a sanity check or when allocating a dynamic buffer, and the number of bytes actually copied into that buffer.

2. Write-what-where or a more constrained write-where condition – a result of inconsistency between the user-mode pointer used during sanity checking (e.g. as a parameter to `ProbeForWrite`) and the one used as destination for an actual memory operation.

3. Read-where condition – a result of insonsitency between the validated and used input pointer, in case that pointer is only used for reading memory and the leaked data can be transferred back into user-mode.

4. Read-garbage condition – a custom name for a situation where the contents of a not entirely initialized kernel buffer are passed down to user mode; caused by having the number of bytes filled in that buffer to be less than its size (the opposite of a buffer overflow).

A majority of the above violations are commonly found and exploited as stand-alone vulnerabilities; specifically, memory corruption issues such as buffer overflows or write-what-where conditions have been subject to detailed analysis in the past. Due to the apparent lower severity, the latter two items have received considerably less attention from the security community; thus, we included some considerations on how the leakage of specific regions of kernel address space has a security impact comparable to the more invasive problems in the *Case study* section.

In the following subsections, we will focus on techniques to improve the odds of reliably stepping through the first phase of race condition exploitation, i.e. winning the race itself.

4.1 Single CPU scenarios

Unlike ten years ago, it is currently rather difficult (if at all possible) to find a personal computer with less than two or four cores on board. The situation is similar in the case of most commercial virtualization services and physical servers, both of which usually offer more than just a single processor to operate on. While it is rather unlikely to find ourselves in a situation where it is required to win a race condition on a single CPU configuration to compromise the security of a modern platform, this section outlines some techniques facilitating one-core exploitation.

As previously discussed, what makes it exceptionally difficult to successfully exploit race conditions on platforms equipped with a single CPU is the lack of true parallelism – we cannot in fact perform any operation "at the same time" as the vulnerable routine, due to the linear way in which the overall environment works. In this situation, the only means to *inject* a different value in memory within the available time window is to have the kernel function preempted in between of the two memory fetches of interest, and get code execution transferred to a controlled user-mode thread before the vulnerable ring-0 code is resumed. As a result, successful exploitation must involve tampering with the decision-making process of the scheduler.

In general, there are three possible reasons for a thread to be preempted by the system scheduler within a specific code block:

1. The block issues an explicit call to `NtYieldExecution` or an equivalent function, thus indicating that the thread has nothing to do at this point in time and the scheduler can choose another thread to execute on the processor.

2. The CPU time slice allocated for the thread exceeds, thus resulting in a forceful transfer of code execution out of the thread back into the kernel-mode scheduler in order to handle other threads in the operating system.

3. The thread issues an I/O operation that requires a long time to complete, e.g. hits a memory area swapped out to disk, requiring the page fault handler to restore the page in physical memory, typically consuming a significant amount of time compared to normal cache-handled references or even RAM fetches.

It is unusual for most potential race condition syscall targets to issue *yield* calls, so we cannot rely on taking advantage of the first option. Performing multiple attempts to win the race in the hope of having the thread preempted in just the right time and transferred to the flipping thread is in fact a viable solution in case of memory corruption violations which only require a single race win to successfully escalate one's privileges in the system, according to experimental data. We have tested an exploit against one of the MS13-016 vulnerabilities discussed later in this paper (CVE-2013-1254) and found that the average time period it takes to win the race inside of a single-core Windows 7 SP1 32-bit guest inside of VirtualBox, hosted by an Intel Xeon W3690 CPU @ 3.46GHz is approximately 110 seconds. After upgrading the VM to four logical CPUs and using a parallelism-aware exploit, we reached an average ratio of 16 wins per second. This implies that running an attack on a single core can be up to 1750 times slower, but is still feasible if the attacker has reasonably unlimited time during which to carry it out, and few wins are sufficient to fully compromise the machine. Various additional techniques can be used to extend the available time window, therefore improving the odds of having the vulnerable routine preempted within the desired block of code. For details, see further sections.

Semi-reliable exploitation of kernel-mode race conditions similar to those addressed in this paper was previously discussed by *twiz* and *sgrakkyu* during their 24C3 presentation titled "From Ring Zero to UID Zero"[35]. The speakers focused on exploitation scenarios when just a single CPU was available, and came up with several interesting ideas to force the kernel to perform an I/O operation while resolving a #PF exception generated upon accessing a swapped-out page backed up by the *pagefile* or another file in the filesystem, e.g. by pushing the target page out of RAM through spraying of the physical memory with large amounts of irrelevant garbage. We haven't found any trivial and fast way to reliably and continuously swap a page of memory into the slow hard drive storage, and we haven't investigated this further. While the concept of

deterministically generating context switches in kernel mode is indeed tempting, there are some fundamental problems related to some of the ideas presented six years ago with regard to how modern computing works today, e.g. it is not very common to see a shared machine with less than 8 or 12 GB of RAM and no user-specific memory usage quotas in place.

While digging further into the specific algorithm of the scheduler on a single-core machine would surely make an interesting research subject, we don't find it to be a direction of much practical value. In situations where utilizing just one processor to carry out an entire attack is a necessity, following the steps discussed in further sections (but running both racing and flipping threads within the one available core) is expected to do the trick for a majority of scenarios you might find yourself in.

4.2 Page boundaries

On a great majority of hardware platforms used today, time windows consisting of several dozens, hundreds or thousands of instructions are more than enough to carry out a successful attack against a race condition vulnerability. However, you can also find instances of double fetch, where respective fetches are separated by only one or two assembly instructions, forcing the attacker to struggle to achieve a reasonable wins per second ratio (or even get a single hit at all). An example of code suffering from an extremely constrained race condition in shown in Listing 7.

Listing 7: A race condition with an attack window of two instructions.

```
MOV EDX, ds:[nt!MmUserProbeAddress]
MOV ECX, ds:[user-controlled address]

MOV EAX, [ECX]
CMP EAX, EDX
JA BAIL-OUT

MOV EAX, [ECX]

...
Operate on "EAX" as a sanitized user-mode pointer
```

In cases like the above, exploitation starts to get tricky even with multiple cores at one's disposal. A time window of just two instructions which don't invoke context switches, don't require interaction with external peripherals, don't trigger any interrupts and are implemented very efficiently by the processor leaves us with just a few cycles during which specific bits of the user-mode value must be flipped. In fact, the only part of the execution flow that we control here and could potentially use in our favor is the user-mode address stored in the `ECX` register.

One technique to significantly delay the completion of an instruction using a user-provided address to fetch more than one byte at a time is to place the word, double word or quad word in between two adjacent memory pages (e.g.

at an address ending with `fff` on systems with 4 KB pages). Both reading from and writing to such misaligned addresses causes the CPU twice the work it normally does, as the operation involves querying the cache for two distinct pages, performing two virtual address translations, fetching data from likely distant locations in the physical memory, and so forth. While it would be intuitive to expect a 2-3x slowdown in the instruction execution time, experimental results show that completion of this corner case can take up to 13 times the time in which a well-aligned value would be fetched. A personal PC powered by an Intel Core i7-3930K CPU and Windows 7 SP1 64-bit was used as the test environment, running the `mem_access` routine shown in Listing 8 with the *iter=1024* parameter 1024 times, and using the minimal timing as the result for the specific offset within a region. The results of the experiment are shown in Figure 5; please note that the tested code does not properly address numerous CPU-level execution optimization techniques found in Sandy Bridge-E, such as out-of-order, superscalar or speculative execution, and so forth. The numbers provided here are specific to the testing methodology used and are primarily provided for the sake of comparison with results yielded by other techniques.

Listing 8: Function used to perform memory access timing measurements.

```
void mem_access(void *addr, uint32_t *t1,
                uint32_t *t2, uint32_t iter) {

  __asm("mov ebx, %0" : "=m"(addr));
  __asm("mov ecx, %0" : "=m"(iter));

  __asm("mfence");
  __asm("rdtsc");

  __asm("mov edi, eax");
  __asm("@@:");
  __asm("mov eax, dword ptr [ebx]");
  __asm("dec ecx");
  __asm("jecxz @end");
  __asm("jmp @@");
  __asm("@end:");

  __asm("mfence");
  __asm("rdtsc");

  __asm("mov ecx, %0" : "=m"(t1));
  __asm("mov [ecx], edi");
  __asm("mov ecx, %0" : "=m"(t2));
  __asm("mov [ecx], eax");
}
```

The average time to access a dword at any of the 3904 in-page and in-cache line locations was around 1.85 cycles (with small fluctuations), while values residing at offsets in between cache lines (which was 64-bytes long for the machine used) took double the time to be fully obtained (around 4 cycles) and 32-bit numbers spanning across two virtual pages were determined to execute for approximately 25 cycles each; all timings were measured using a standard memory

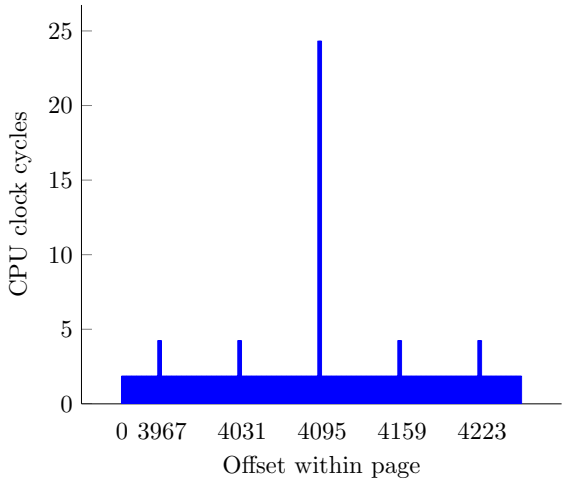

Figure 5: Cycle cost of 32-bit memory references at respective offsets within a cached region.

allocation after ensuring that each of the tested regions was present in the L1 cache prior to the test. Considering that the `CMP` and `JA` instructions typically consume around a single cycle on the same test machine, introducing the above "optimization" resulted in a roughly 2500% time window growth.

While the observation can be successfully used to extend the available time window by slowing down instructions in kernel-mode, it doesn't necessarily have to same effect on the *flipping threads*. In scenarios where the attacker desires to use the time window to increase the raced value in memory, he can take advantage of the *little endian* ordering used in the x86 architecture, causing the more significant bits of the number to be stored at higher addresses. For example, if the target value is a 32-bit pointer spanning across two pages with two bytes in each of them, the malicious threads can cause the full 32-bit address to continuously switch between user and kernel-mode by persistently *xoring* the upper 16 bits of the value at the base of the second page against 0x8000. Making use of the page boundary slowdown by employing two CPUs is better illustrated in Figure 6.

While page boundaries appear to be most useful in constrained scenarios where the memory address used is the only part of the context directly controlled by the attacker, they can also prove functional in *easier* instances of race condition bugs. In particular, it is worth to keep in mind that for a time window consisting of n instructions fetching from independent user-mode memory regions, each of them can be used similarly to maximize the reliability of our exploit.

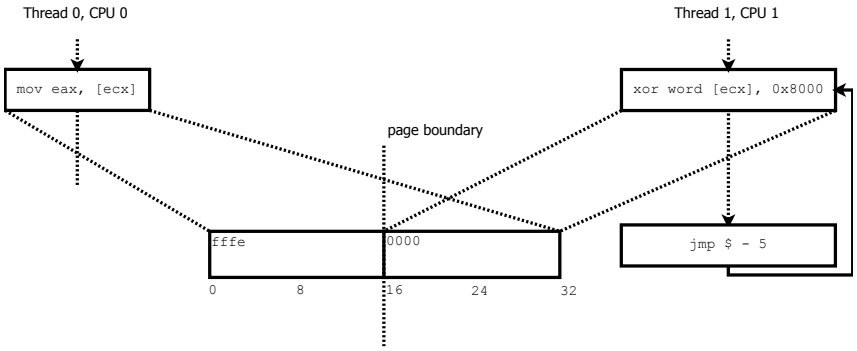

Figure 6: The optimal exploitation scheme involving a page boundary.

4.3 Memory caching and write-combining

Experimental timing results in the previous section were highly affected by the fact that all memory locations referenced during the benchmark were ensured to be found in the CPU L2 cache. The fetch timings could have been further increased if instead of prefetching the data into cache, we would do the opposite – make sure that the cache buffer is filled with junk throughout the test. However, we have found that instead of playing with internal CPU optimization structures, it is far more effective to disable cache entirely for pages used in the attack, forcing the processor to interact with the physical memory controller in a synchronous manner, until the desired value is fully read / written to the memory dice.

Non-cacheable private memory can be allocated by any user-mode application in the Windows environment by using the `PAGE_NOCACHE` flag as a parameter to the `VirtualAlloc` function. Internally, information about page non-cacheability is encoded within one of the eight *Page Attribute Table* entries (see *Intel 64 and IA-32 Architectures Software Developer's Manual*, *Volume 3A: System Programming Guide, Part 1*, Section *11.12 PAGE ATTRIBUTE TABLE (PAT)*), pointed to by the `PAT`, `PCD` and `PWT` bits the page table entry for the specific memory region.

When we prevent the CPU cache from being used to boost up our memory references, page boundaries no longer matter. As we communicate directly with RAM to acquire the interesting memory contents, we now operate on the granularity of (much smaller) physical memory blocks used by the underlying hardware configuration. The size of a single physical memory block size on the desktop PC used for previous tests was experimentally determined to be 64 bytes. On the other hand, the size of such blocks on a notebook driven by AMD Turion 64 X2 TL-56 and two dices of DDR2 memory was only eight bytes. The memory access timings for subsequent offsets within a non-cached page are shown in Figures 7 and 8 for both discussed machines.

Based on the charts, we can draw two important conclusions. First, the

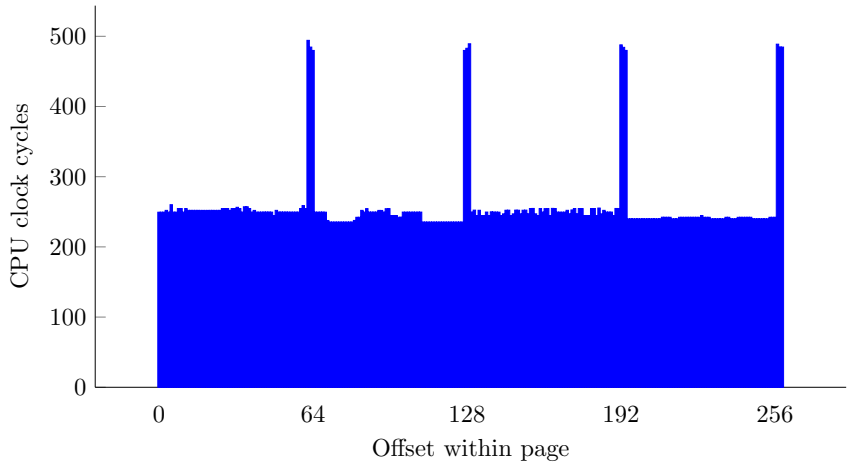

Figure 7: Cycle cost of 32-bit memory references at respective offsets within a non-cached region, block size = 64 bytes.

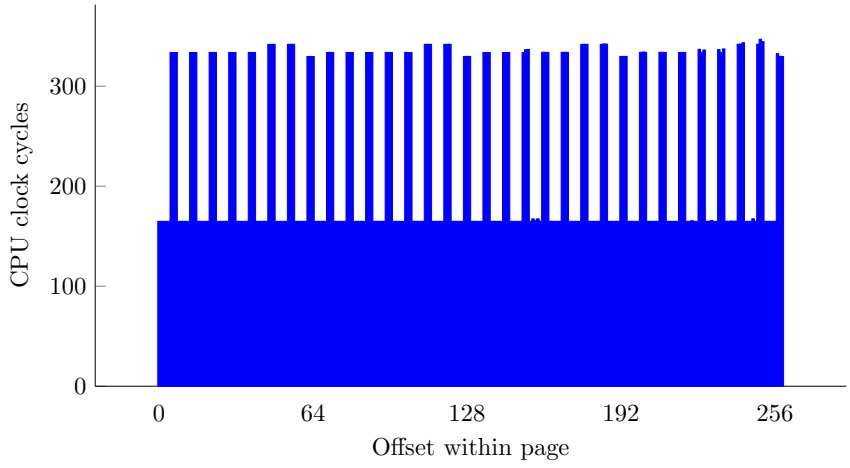

Figure 8: Cycle cost of 32-bit memory references at respective offsets within a non-cached region, block size = 8 bytes.

cost of fetching a 32-bit value from a properly aligned address on the test PC is typically around 250 cycles, which is ~13000% the cost of fetching the same piece of memory from cache (~1.85 cycles) and ~1000% the cost of fetching it from a boundary of two cache-enabled pages (~25 cycles). This gives an attacker an enormous edge while trying to win a race against such instructions. The second important observation is that similarly to page boundaries, references to memory at block boundaries also take much longer that they normally would. Indeed, it is clearly visible that two separate blocks of memory must be acquired from RAM instead of one, as the timing relation between aligned and misaligned accesses is almost exactly $\frac{1}{2}$.

Given all the above, the most sensible option is to use a non-cached, misaligned memory address when confronted with a constrained vulnerability which doesn't provide us with other choices as to how to slow the kernel down and win the race. As the physical memory block sizes can differ from target to target, it is safe to stick with page boundaries, granted that the page size on any x86 and Windows driven machine will always be divisible by the block size.

It is also worth to mention the existence of another page attribute of potential interest – write-combining – supported both by Intel and the Windows API interface (see `PAGE_WRITECOMBINE`). The attribute was designed specifically for interacting with external devices having some of the virtual address space mapped directly to their internal memory (such as VRAM), and implemented in order to reduce the volume of back-and-forth communication with that device by *packing* large chunks of continuous read or write operations into single packets. Due to the nature of the data going through write-combine mapped memory, it is not subject to caching – therefore, it can be used interchangeably with the `NO_CACHE` flag during race condition attacks. Our tests have shown that neither attribute is particularly more effective than the other, as they both result in roughly same access timings. It is possible that some specific characteristics of how x86 processors implement the write-combining mode could be used in the attacker's favor; however, we haven't further investigated the feature.

4.4 TLB Flushing

The process of fetching data from virtual memory consists of several major stages, some of which can be manipulated in non-trivial ways. While forcing the CPU to double the required work by reading values from across two memory pages and putting them directly in RAM provides us with a significant advantage at virtually no cost, there are still portions of the process that can be influenced by a user-mode application. Before a processor can read the actual bytes from either cache or physical memory, it must *know* where to look for them. Code running in *Protected* or *Long Mode* uses an additional layer of memory addressing called *virtual addressing*, causing the processor to perform address space translation in order to obtain a physical address of the region referenced in assembly code. As the structures describing current virtual address space reside in RAM and therefore are very expensive to read, all x86-family processors implement a special address translation cache called "Translation

Lookaside Buffer". As TLBs usually cover most or all of the pages referenced by a thread within its time slice, the performance cost of address translation is usually disregarded, being negligibly small (within a few cycles). As you can imagine, user-mode threads in Windows do have indirect control over the contents of TLB, and even better, they can remove specific entries out of the translation cache.

Mangling with the internal state of TLB is beyond the scope of the paper; however, if you are interested in cache-related security considerations, refer to the "Practical Timing Side Channel Attacks Against Kernel Space ASLR" paper[29]. Flushing the lookaside buffer, on the other hand, is extremely easy, as it can be achieved with a documented Windows API `EmptyWorkingSet` function. The concept of *process working sets* describes the set of pages mapped within the virtual address space of a particular process which are found in physical memory at a given time (i.e. are not swapped out). The aforementioned API allows processes to instruct Windows to remove as many pages from a process working set as possible, consequently freeing up physical memory for other applications to use (in a sense, this functionality is the opposite of the virtual memory locking mechanism available through `VirtualLock` and `VirtualUnlock`). As the documentation states[26], pages which do not belong to at least one working set are not automatically removed from RAM, but rather sit there until the system needs to free up some resources. As a result, calling the `EmptyWorkingSet` gives no guarantees of which or whether any pages are indeed going to be transferred to a hard drive – what it does guarantee, though, is that the TLB entries for the swapped out pages, or potentially the entire TLB cache altogether is going to be flushed. A similar (but more directed) effect can be achieved by calling the `VirtualUnlock` function against a non-locked memory region, as the documentation states:

> Calling VirtualUnlock on a range of memory that is not locked releases the pages from the process's working set.

The interesting property of removing all or a specific entry from TLB is that every first access to a memory page no longer found in the cache forces the CPU to perform a complete *page walk*, that is traverse the Page Table structures in RAM in search of the specific page entry, in addition to the normal fetching of value from memory. On 64-bit platforms where a typical Page Table consists of four layers, this adds up to four separate reads from RAM, plus the one for the value itself; if attacking *Protected Mode*, there is one read less[15]. Note that while some of the Page Table entries may be longer than the size of the native CPU word, each fetch from physical memory obtains data of the RAM native block size (always observed to be at least 8 bytes), which is sufficient to read an entire Page Directory or Page Table entry.

A comparison of experimentally acquired memory access timings of reads from cached memory, non-cacheable memory and non-cacheable memory with

[15]The exact number of Page Table layers can be different depending on the specific configuration of the victim machine (e.g. if the Physical Address Extension or Large Pages are enabled); in reality, the number can vary from one to four layers.

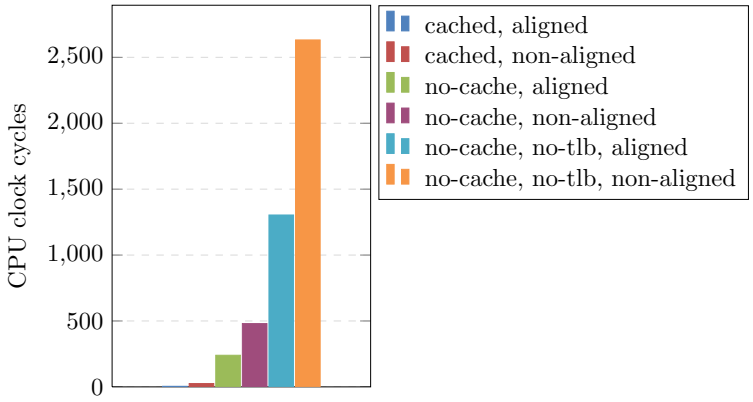

Figure 9: Cycle costs of 32-bit memory references with different attributes applied to the memory region and its address.

prior TLB flush performed is presented in Figure 9. It is clearly visible how the "no-tlb" memory operation requires roughly five times the cycle count of a single RAM read performed during the second test. Also, using a page boundary turns out to be once again quite helpful, as it requires two page walks to be done instead of just one. While this process could be optimized as the Page Table paths usually greatly overlap for two adjacent pages, it is apparent that the tested Core i7-3930K CPU doesn't implement special handling of this corner case.

While invalidating the TLB can give an attacker a significant edge in specific cases, it is in fact much more *expensive* than the other techniques explained so far. Placing an address in between two pages requires a single pre-calculation; marking memory as non-cacheable only takes a single `VirtualProtect` call – on the other hand, ensuring empty TLB needs either one of the `SetProcessWorkingSetSize`, `EmptyWorkingSet` or `VirtualUnlock` functions to be used before each call to the vulnerable kernel routine, consequently decreasing the total number of races we can attempt per second. Below follow the calculations of at which point using the technique can increase the total number of cycles comprising the time window per second.

Let c be the CPU clock frequency, w the length of the original time window in cycles, e the number of extra window cycles provided by the TLB flush, x the cycle cost of a single race attempt (i.e. invoking the vulnerable syscall) and t the additional time incurred by the TLB-flushing system service. Assuming that we have the CPU entirely for ourselves, the total time window size per second in a normal scenario can be expressed as:

$$\frac{w}{x} \times \frac{c}{x} = \frac{wc}{x} \tag{1}$$

With TLB flushing, both the time window and total time spent in kernel mode per attempt increase, therefore changing the time window per second to:

$$\frac{w+e}{t+x} \times \frac{c}{t+x} = \frac{(w+e)c}{t+x} \tag{2}$$

In order for the technique to be beneficial during a practical attack, the (2) > (1) condition must be met:

$$\begin{aligned} \frac{(w+e)c}{t+x} &> \frac{wc}{x} \\ (w+e)cx &> wc(t+x) \\ cwx + cex &> ctw + cwx \\ ex &> tw \\ \frac{e}{t} &> \frac{w}{x} \end{aligned}$$

The result is rather intuitive – the $\frac{\text{window size increase}}{\text{additional cost}}$ ratio must be greater than $\frac{\text{existing time window size}}{\text{existing time cost}}$. Before making a decision on whether the technique is beneficial for the attack in your specific scenario, one should carefully measure all timings found in the equation in order to determine if the overall time window per second grows or decreases with the change.

4.5 Thread management

The techniques and observations discussed in the previous sections allow an attacker to maximize the extent of the race time window; attributing the correct number and types of threads to CPUs available on the target machine ensures that the time window (however big or small) is being raced against in the most optimal way. Properly distributing work across the cores is essential to effective exploitation.

The single-core scenario has already been addressed – from this point forward, let's assume that we can operate on n logical CPUs, $n \geq 2$. The first and foremost observation to emphasize is that since multiple logical CPUs are involved in the process, it is no longer required to trigger as many context switches as possible, but rather the opposite – make sure that as much CPU time is spent executing our payload instead of the operating system thread scheduler. When no CPU quotas are set up in the system, any thread executing an infinite loop with no "yield" requests in between will by default receive the maximum share of CPU time that Windows can afford to assign to the thread. Therefore, the first rule of thumb for local race condition exploitation is to only run a single thread on every available logical CPU.

At this point, we need to make four further, important decisions: which thread type (racing or flipping) should be scheduled for each CPU, which or how many distinct memory areas should be targeted by these threads, what attributes are to be assigned to the memory pages and finally, if there are any

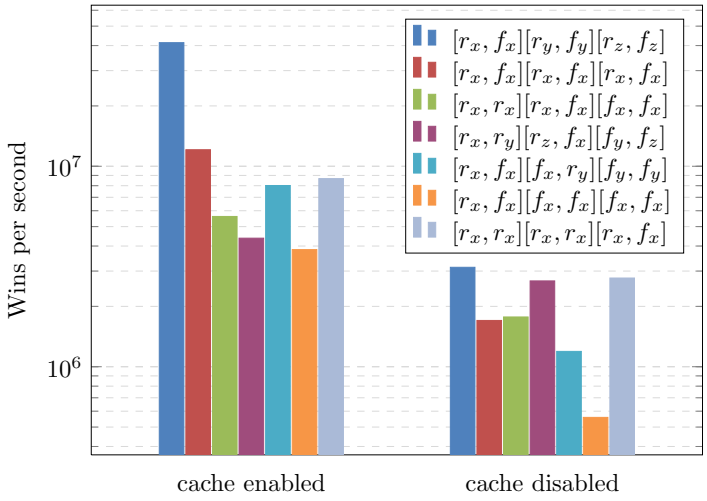

Figure 10: Number of race condition wins for each threat assignment strategy, on six Hyper-Threading enabled cores (three physical cores).

variations of the different settings that could further improve the exploitation reliability. We are going to approach each of the decisions separately.

We have performed a number of performance tests using different amounts of racing / flipping threads assigned to different cores and targeting different memory areas. The tests were conducted by using a custom application which simulated the presence of a race condition vulnerability and counted the volume of wins per second for each of the tested thread assignment strategy. While running the tests, an interesting fact came up – it makes an enormous difference whether two logical processors used for exploitation are on the same chip or core (due to the Hyper-Threading technology) or not. As the next step, we chose a representative set of results and presented them in Figures 10 and 11 for HT-enabled and physical separate processors, respectively. We used the same testing machine equipped with an Intel Core i7-3930K CPU. Each tested configuration should be read as follows: racing (i.e. invoking the vulnerable code path) threads are denoted as r, flipping threads as f. The x, y and z subscripts indicate different memory regions, e.g. where y and z are not used, all threads were set to target a single memory location. In the former chart, logical CPUs running within the same physical cores are grouped together in the legend.

Please note that the specific numbers obtained during the experiments can, and will be different on any other software or hardware configuration you might try – your mileage may vary. However, the relations in performance between each pair of assignment strategies are believed to hold true for most platforms seen today.

There are several far-reaching observations that can be inferred from the

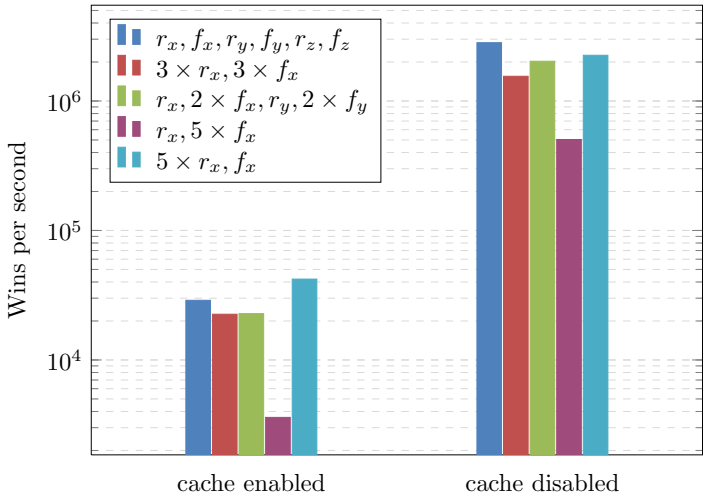

Figure 11: Number of race condition wins for each threat assignment strategy, on six separate physical cores.

two charts. Most importantly, splitting all available processors into pairs of $(race, flip)$, each targeting a different memory region appears to generate by far the best results out of all tested options, both with and without HT enabled. Secondly, two logical CPUs in a single physical chip racing against each other reach the highest race condition win rates when working with cacheable memory. On the other hand, in case of six completely separate cores (HT disabled), it is conversely most effective to target memory pages marked as non-cacheable and therefore stored directly in RAM at all times.

We believe that the Hyper-Threading phenomenon can be explained as follows: each two logical CPUs on one physical core are most effective (performance-wise, not exploitation-wise) when executing unrelated operations within different regions of memory and therefore not blocking on each other. When running two threads that persistently fetch and write to the very same memory cell, the instructions cannot be ran in parallel, but instead have to wait until each of them complete. In consequence, this leads to ideal exploitation conditions – code belonging to each of the threads start interlacing very frequently, generating tens of millions of race wins per second. Because accessing the CPU cache can be dozens of times faster than operating on physical memory, using cacheable pages for the attack shortens the execution time of each racing / flipping instruction, resulting in having more instructions execute – and more interlacing occur – within one second. To conclude, whenever a HT-enabled platform is found to be the target of an attack, the optimal strategy would be to create $\frac{n}{2}$ pairs of threads for n logical CPUs, each consisting of a *race* and *flip* type thread and targeting a distinct region of cached memory.

In scenarios where Hyper-Threading doesn't come into play, and we are limited to physically separate cores, the optimal solution is largely different. As we saw in Figure 11, race condition exploitation is most efficient when using non-cacheable memory, resulting in each memory operation taking relatively long to complete, therefore enabling us to win more races over time. However, it is not really the goal to make both *fetch* and *flip* operations as slow as possible – in fact, this is only desired for the *fetch* part, as the best win rates should be presumably achieved with a possibly slow racing thread and possibly fast flipping thread. This opens up room for some combinations of the already known techniques to prove effective, such as page boundaries with different attributes set for the two adjacent pages.

Figure 12 presents the comparison of race win rates for six different sets of characteristics for the 32-bit area under attack, measured on six HT-disabled CPUs. The configurations cover scenarios with the dword spanning across one or two pages, with the first or both of them marked as non-cacheable or write-combining. Where the variable crosses a page boundary (tests 3-6), the flipping thread would only modify the 16 most significant bits residing within the second memory page. Based on the results, it is clearly visible that for non-HT configurations, it is most effective to place the raced value in between a non-cacheable and a cacheable page respectively, and only flip bits within the cached part of the value. Similarly to Hyper-Threading scenarios, each such memory region should be assigned to a separate pair of threads, one triggering the vulnerable code path and the other doing the flipping work. This configuration is believed to be the most effective one for typical desktop PC configurations where NUMA and other advanced memory management technologies are not involved.

Additionally, the chart further confirms that there are no significant differences in choosing between the `PAGE_NOCACHE` and `PAGE_WRITECOMBINE` allocation flags, as they both achieve similar performance results.

4.6 Flipping operations

The type of assumption violated by the vulnerable kernel routine largely determines the type of the operation which should be ideally performed by the *flipping thread*. If successful exploitation depends on a binary decision (i.e. successful pass through `ProbeForWrite`), the *xor* operation works really well, as it can be used to *automatically* switch between the "safe" and "unsafe" value of a variable. In case of a pointer, the constant xor operand could be 0x80000000, in which case the address would continuously switch between user and kernel-mode. In other cases, it might be enough to just have the raced value to be *different* from what it was a while ago, without much control of what the original and modified numbers actually are – an example of such scenario would be the allocation and filling of a heap-based buffer based on a double-fetched value. In such situation, it would be likely sufficient to indefinitely increment the variable in question, hoping that at one point the number of copied bytes would exceed the allocation size.

While the xor operation provides better precision (if a constant operand is

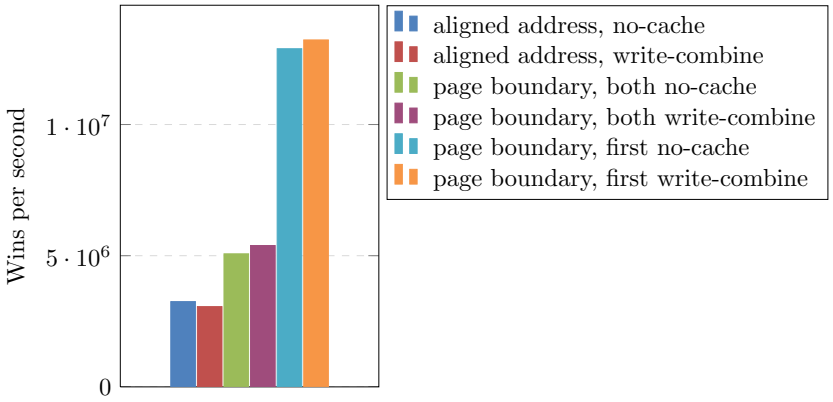

Figure 12: Number of race condition wins for different page attribute and offset variations, on six separate physical cores.

used), it comes at a performance cost: the race is only won if the binary instruction executes an odd number of times within the time window, whereas for arithmetic instructions such as addition or subtraction, it is enough to execute them any non-zero number of times. As a result, the usage of arithmetic operations is approximately two times faster than xor but leaves the attacker uncertain of how exactly the value changes (and thus, for example, how many bytes are overwritten in a kernel buffer). When deciding on the specific operation to use in your exploit, please consider if precision or the race winning rate is a priority in your particular use-case.

4.7 Priority classes

Each thread in a Windows session is subject to thread scheduling implemented by the operating system. As some tasks are more time-critical than others or require more computation power to complete, one property assigned to every thread is its *kernel priority*, calculated as a function of the *process api priority* and *thread api priority*, two values controlled through the `SetPriorityClass` and `SetThreadPriority` API functions. By raising the process/thread priorities to `HIGH_PRIORITY_CLASS` and `THREAD_PRIORITY_HIGHEST` respectively (these are the highest classes available to every user in the operating system), an exploit can make sure that it would not be preempted by normal-priority tasks running in the context of the attacker's and other users' accounts. Whether mangling with the priorities makes much sense in a practical attack depends on the type of the target machine – if it is under heavy load, has multiple threads running in parallel to the exploit, and so forth. In our typical testing environment with no other processes competing for CPU time, changing the priority class has hardly any effect on the exploit performance; since it is

difficult to simulate the execution environment of a Windows server workstation with multiple users logged in, no further tests were performed in this area.

With regards to the task scheduling algorithm implemented in Windows, the two obligatory reads are section *"Processes, Threads and Jobs: Thread Scheduling"* in Windows Internals 6. Part 1[12] and the "Scheduling" section at MSDN[25].

5 Case study

Throughout several iterations of running Bochspwn against Windows 7 and Windows 8 in both 32 and 64-bit versions, the project has identified around 100 unique instances of multiple user-mode memory fetches reported to Microsoft. Although most of the reports were turned down as either non-exploitable or requiring administrative rights for successful exploitation, many were actual security vulnerabilities. In order to develop and test the race condition exploitation techniques discussed above, as well as prove the feasibility of using the bugs in real-life scenarios, we have created fully functional exploits for two of Bochspwn's findings, each slightly different in terms of the attacker's gains, yet both of them eventually leading to an elevation of privileges within the operating system. Each subsequent subsection aims to fully explain the nature of the security issue in question, how the race can be won within the time frame existing in the vulnerability and how the resulting violation could be used to a hostile user's benefit.

While actual proof of concept source code is not explicitly provided with the paper, we encourage the reader to reproduce the described steps and try to reconstruct working exploits from the available information.

5.1 CVE-2013-1254

Let's begin with a vulnerability – or, in fact, a family of 27 distinct vulnerabilities – which is particularly special for the Bochspwn project. The overall group of bugs, otherwise identified as CVE-2013-1248 to CVE-2013-1250 and CVE-2013-1254 to CVE-2013-1277 was originally found by j00ru several years back (likely around 2010 / 2011), but ignored or forgotten at the time. Several years later, in August 2012, he accidentally stumbled upon these bugs again while doing unrelated research, and decided to investigate them in more detail. As it turned out after several days, these bugs were real and exploitable, thus we reported them to Microsoft via the usual means. Due to the fact that all flaws were caused by the very same faulty code pattern, we started considering different approaches to find more issues of the same kind. None of these bugs were originally Bochspwn's findings, but they inspired the overall research and were successfully used as a control group, in order to confirm that our implementation worked correctly.

In order to understand why the vulnerabilities existed in the first place, and how they were likely fixed with a single change in a shared macro definition,

some background on the inner workings of `win32k.sys` is required. This is covered in the next section.

5.1.1 The vulnerability

Although the mechanism of user-mode callbacks is an obscure, internal detail strongly tied to the overall Windows GUI architecture and not particularly useful to anyone but core Microsoft kernel developers and possibly some security professionals, it has been publicly discussed in great detail; specifically because of the fact that its incorrect usage has led to a number of serious privilege escalation vulnerabilities in the operating system (see "Kernel Attacks through User-Mode Callbacks"[33] and "Analyzing local privilege escalations in win32k"[27]), and its implementation strongly affected the exploitability of seemingly unrelated issues (see "The story of CVE-2011-2018 exploitation"[14]). For the interesting low-level details related to the mechanism, please refer to the three aforementioned papers.

The general idea behind the feature is as follows: because of the client/server design of GUI interactions (i.e. a request to perform an operation issued by one process may concern a graphical element managed by another process), many system call handlers and nested routines called within them must *consult* the user-mode portion of a process (the `WndProc` routine, or one of the built-in `user32.dll` functions) before proceeding with an operation. Depending on the specific window event, these callbacks are used to inform user-mode (pass data), ask user-mode (acquire more data) or both. A simplified illustration of the execution flow is presented in Figure 13.

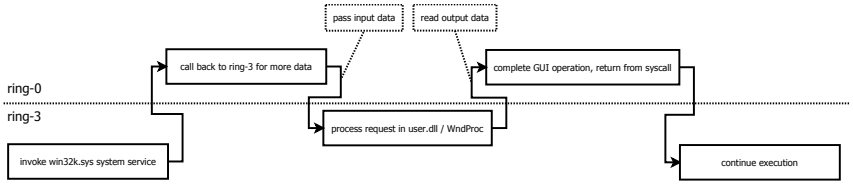

Figure 13: Execution flow of a `user` syscall involving a user-mode callback.

Internally, output data from user-mode is passed back to kernel-mode through a trivial structure presented in Listing 9 (the structure definition is not officially available, but given its small size, the semantics of each field can be easily inferred by reverse engineering relevant code). The structure stores information about the exit code of the user-mode handler invoked upon the callback, and a pointer / length pair specifying where in ring-3 memory the actual data can be found. A pointer to the structure is traditionally passed as the first parameter to the `NtCallbackReturn` service, responsible for restoring the previous execution context from information stored on the kernel stack and returning from the original `KeUserModeCallback` call.

Listing 9: Reverse-engineered definition of the output callback structure.

```
typedef struct _CALLBACK_OUTPUT {
  NTSTATUS st;
  DWORD cbOutput;
  PVOID pOutput;
} CALLBACK_OUTPUT;
```

After the callback returns, it is up to the `win32k` caller to process the output pointer in `CALLBACK_OUTPUT`, which typically takes the form of following the steps:

1. Check that the output data length is 12 bytes (size of the structure).
2. Check that the structure resides within user-mode.
3. Check that `CALLBACK_OUTPUT->pOutput` resides within user-mode.
4. Copy n bytes from `CALLBACK_OUTPUT->pOutput` into a local kernel buffer, where n is specific to the caller function (usually between 4 and 128 bytes).
5. Further operate on the locally saved copy.

While each particular step is fundamentally simple, you can notice that there is potential for a multi-fetch condition here – the steps include referencing a user-mode value for sanity checking and using it as the `src` parameter to inlined `memcpy`. This is by no means an unusual situation, as the kernel continuously fetches, verifies and uses pointers residing within user-mode; however in this particular case, the disassembly for the relevant part of the code looked as shown in Listing 10. In the assembly snippet, the `ECX` register would point into a user-controlled `CALLBACK_OUTPUT` structure. As the listing clearly illustrates, there are two subsequent user-mode fetches: one performed implicitly by the `CMP` instruction and the other one executed if the preceeding sanity condition is met, separated by a single conditional branch. Further in the code, the value obtained in the second read is used as the source address to copy 28 bytes (which is the case for `win32k!ClientGetMessageMPH`, used as an example). Without doubt, this is an obvious instance of a *time of check to time of use* race condition with both user-mode memory reads placed almost directly one after another.

Listing 10: Flawed handling of the CALLBACK_OUTPUT structure.

```
.text:BF8C4505    mov     eax, _W32UserProbeAddress
                  .
                  .
                  .
.text:BF8C4524    cmp     [ecx+CALLBACK_OUTPUT.pOutput], eax
.text:BF8C4527    jnb     short invalid_ptr
.text:BF8C4529    mov     eax, [ecx+CALLBACK_OUTPUT.pOutput]
.text:BF8C4529
.text:BF8C452C invalid_ptr:
.text:BF8C452C    push    7
```

```
.text:BF8C452E    pop     ecx
.text:BF8C452F    mov     esi, eax
.text:BF8C4531    rep movsd
```

After identifying the problem in one routine, we found that identical three-instruction assembly patterns are also present in 26 other `win32k.sys` functions, "incidentally" always around a call to `KeUserModeCallback`, thus in fact representing the exact same type of vulnerability. A list of functions determined to be affected by the problem is shown below; it is possible that more instances were found and patched by Microsoft internally in the process of variant analysis.

```
CalcOutputStringSize
ClientGetListboxString
ClientGetMessageMPH
ClientImmLoadLayout
CopyOutputString
SfnINOUTDRAG
SfnINOUTLPMEASUREITEMSTRUCT
SfnINOUTLPPOINT5
SfnINOUTLPRECT
SfnINOUTLPSCROLLINFO
SfnINOUTLPUAHMEASUREMENUITEM
SfnINOUTLPWINDOWPOS
SfnINOUTNEXTMENU
SfnINOUTSTYLECHANGE
SfnOUTLPCOMBOBOXINFO
SfnOUTLPRECT
SfnOUTLPSCROLLBARINFO
SfnOUTLPTITLEBARINFOEX
fnHkINDWORD
fnHkINLPCBTCREATESTRUCT
fnHkINLPMOUSEHOOKSTRUCTEX
fnHkINLPRECT
fnHkOPTINLPEVENTMSG
xxxClientCopyDDEIn1
xxxClientCopyDDEOut1
xxxClientGetCharsetInfo
xxxClientGetDDEHookData
```

The general scheme was always similar in each affected function – start with a call into user-mode, then perform basic sanitization over the `CALLBACK_OUTPUT` structure and finally use the `pOutput` field found at offset 8 to retrieve the contents of a function-specific structure. After careful analysis of the compiler-generated code found in those functions, we have a strong feeling that the vulnerability was likely a result of using a macro similar to the following:

```
#define FETCH_USERMODE_STRUCT(x, type) ((x) < MmUserProbeAddress ?
                                        *(type *)(x) :
                                        MmUserProbeAddress)
```

used in the following manner:

```
LocalStructure = FETCH_USERMODE_STRUCT(CallbackOutput->pOutput,
                                       STRUCT_TYPE);
```

Although it is a wild and unconfirmed guess, there are several arguments to support the claim:

1. The very same assembly code is shared across all affected routines with a 1:1 opcode bytes relation, implying that exactly the same high-level code pattern was used to generate each of those code snippets.
2. The overall assembly construct is highly optimized, which is especially characteristic to brief *inline if* C statements such as the one presented above.
3. The structure-copying code which follows pointer sanitization is highly optimized for size. For relatively long structures, an inlined implementation of `memcpy` is used as illustrated in Listing 10; shorter ones are copied using chained `MOV` instructions (see `SfnINOUTSTYLECHANGE` for an example). Such optimizations are specific to direct structure assignments used in C source code.

Based on what we know about the vulnerability at this point, several observations come to mind. First of all, the value fetched twice in a row is always used exclusively in a *read* operation, i.e. while copying a final structure from user-mode; the address is never written to. The fact renders any kind of kernel memory corruption impossible, greatly limiting potential vectors of exploitation. Instead, if an attacker was able to provoke inconsistency in the referenced memory region so that the value obtained during the second fetch was arbitrary, the vulnerability would make it possible to get the kernel to read data from ring-0 address space and possibly leak the data back to user-mode. In other words, each of those 27 issues could be used (at maximum) to read arbitrary kernel memory or crash the operating system by triggerring an unhandled *access violation* exception if the race is won, which might be a difficult task in itself due to the narrow time window available.

5.1.2 Winning the race

Considering the fact that the available time window is limited to a single conditional branch and a portion of the preceding memory-fetch instruction, winning the race a reasonable number of times per second requires the usage of a majority of techniques explained in the paper – specifically, placing the dword subject to the race at a page boundary, with the first page being marked as *non-cacheable*, flushing the TLB each time prior to invoking the vulnerable code path, using an optimal thread assignment strategy (depending on the hardware configuration) and using *xor* as the flipping operation. By making use of the above tricks, we have been able to achieve a result of up to 28 kernel memory reads per second on a 5-year old laptop equipped with a AMD Turion 64 X2 TL-56 1.6GHz CPU.

It should be noted that in addition to extending the attack window itself, one can also employ techniques purposed to increase the frequency of the vulnerable code path execution. For example, it is possible to reduce the volume of unnecessary instructions by replacing a call to the documented `SetWindowLong` API function triggering the bug with a direct `NtUserSetWindowLong` system service invocation. Generally, while the execution of kernel-mode instructions comprising the implementation of the vulnerable routine usually cannot be avoided, the extent of code executed in user-mode (in between system calls) is fully controlled, and should be reduced to a minimum for best attack efficiency.

5.1.3 Leaking kernel memory

Having the ability to read arbitrary kernel memory with a reasonable frequency doesn't automatically grant an attacker elevated privileges in the operating system, but opens up room for different types of attack. Unlike typical memory corruption issues where the problem is *what* and *how* to overwrite in memory to hijack the execution flow while maintaining system reliability, the question here is different – what secret information can be found in the kernel address space that can be effectively used to escalate one's privileges. As it turns out, the problem is not trivial.

To our best knowledge, Windows in the default configuration does not explicitly and purposely store information which could be directly used to impersonate another user, or otherwise gain any superpowers in the system. For example, the overall user authentication mechanism is implemented in a user-mode `lsass.exe` process, while the kernel itself doesn't store plain-text passwords. As passwords are the only true secrets making it possible to elevate user privileges, we are forced to look for other types of data which could be indirectly used for the same purpose. Several ideas come to mind.

First of all, several exploit mitigations implemented in the latest builds of the NT kernel are based on pseudo-random, secret values by design unknown to an attacker. One example of such mechanism is *Address Space Layout Randomization* (ASLR), which ensures that executable images, heap structures and other portions of data are randomly relocated, therefore making it difficult or impossible for a hostile user to predict the address space layout, often significantly hindering the exploitation process. Considering the regular structure and predictable address of the *Page Directory* and *Page Table Entries* themselves (e.g. addresses `c0000000` and `c0600000` in 32-bit PAE-enabled platforms), it would be easily possible to use the race condition to fully disclose the address space layout and thus possibly facilitate the exploitation of another kernel vulnerability. Realistically, this is not particularly useful, as both the x86 architecture (as implemented by Intel and AMD) and Windows have been shown to be affected by shortcomings that make it possible to disclose the information in a plethora of other ways[29] [15].

Another mitigation relying solely on the secrecy of a value stored within kernel-mode are *stack cookies*, commonly known as "GS cookies" in Microsoft compilers. Stack cookies are designed to prevent exploitation of continuous

stack-based buffer overruns by verifying the consistency of a cookie against the expected value stored in static memory of a specific device driver, prior to returning from a function. Although several practical attacks have been developed against the cookie generation mechanism found in Windows 7 and prior versions[18], disclosing the exact value of the cookie (stored under `__security_cookie`) would make it possible to fully and reliably evade the mitigation, thus the race condition would be a perfect fit when combined with a stack-based overflow, such as CVE-2010-4398.

Furthermore, all commonly used transparent software disk encryption solutions tend to store the disk decryption key in plaintext in kernel memory, as the decryption itself takes place in a device driver. The fact that the decryption keys are found in physical memory has been used to demonstrate successful *Cold Boot Attacks* against TrueCrypt[8]; similarly, they could be read by a user-mode application from the kernel address space directly. While obtaining the keys themselves doesn't reveal the volume password and generally doesn't allow for elevation of privileges in the system, it could be very well used in combination with a physical attack, where the attacker later obtains a raw dump of the disk.

Thinking of remote scenarios, it is important to note that all user-mode system calls are internally handled by the kernel, including interactions with the system registry, or external peripherals (network card, hard drive etc.). When handling those requests, the underlying device drivers often dynamically allocate helper buffers to store the requested data before copying it back to the requestor's user address space. Because none of the allocations are overwritten while being freed, and some of them are not immediately reused for other purposes, there is a large number of memory chunks containing potentially sensitive data just floating around in the kernel memory. By "scanning" the ring-0 address space in search of interesting pieces of files, network communications or registry activity, an attacker could possibly identify plain-text passwords, bank account pins, web browser cookies used for authorization and a number of other potentially dangerous types of information belonging to other users who share the target machine. However, considering that the user-mode callback race conditions would only allow an attacker to leak optimistically up to 4kB of data per second, it is unlikely that randomly scanning the kernel space at this pace would yield interesting results – in order to be effective, the discussed vulnerability requires the attacker to know precisely which memory area contains useful information.

One such specific type of information which can be found in the kernel structures are NTLM hashes of all users in the system. The hashes of users' account passwords are traditionally stored in the `HKLM\SAM\SAM\Domains\Account\Users\?\V` registry values, which have been experimentally determined to be cached by the operating system in memory, thus persistently residing in the kernel address space. Although officially undocumented, the *Config Manager* (an internal name for the registry kernel subsystem) structures are available for analysis via Microsoft Symbol Server, and they are generally easy to understand and traverse. Extracting the password hashes is believed to be the closest

one can get to obtaining explicit authorization data, and should be enough to escalate one's privileges in cases where weak passwords are used.

Finally, an attacker could simply target input devices such as the keyboard or mouse. All physically attached HIDs are handled by respective device drivers, which must store the data coming from external interrupts in kernel memory at one point in time, in order for the system core to propagate the event in the executive environment. Thanks to the fact, it is possible to sniff on a physical user's keyboard and mouse which in result is very likely to eventually disclose that user's password in plain text, ready to be used for successful impersonation by the exploit. This exploitation scenario will be discussed in detail in the next section; while we assume that the victim administrator uses a physically attached PS2 keyboard to interact with the system, the principle should also apply to other sockets (e.g. modern USB keyboards) or even remote terminal connections.

5.1.4 Sniffing a PS/2 keyboard

In Windows, keyboard and mouse devices connected through PS/2 have corresponding interrupts, both handled by the `i8042prt.sys` driver. In WinDbg, it is easy to locate both interrupts using the `!idt /a` command with kernel symbols loaded:

```
kd> !idt

Dumping IDT:

  ...
61:        85a4d558 i8042prt!I8042MouseInterruptService (KINTERRUPT 85a4d500)
  ...
71:        85a4d7d8 i8042prt!I8042KeyboardInterruptService (KINTERRUPT 85a4d780)
  ...
```

The process is similarly simple when performed programatically within the guest system: the IDT base address for the current CPU (important – make sure the thread is pinned to a single core at this stage) can be obtained using the `SIDT` instruction. Furthermore, we can read the entire table of 256×8 bytes, which gives us a full picture of the interrupt handlers registered in the system. Out of each 8-byte structure, we can extract a 32-bit pointer by concatenating the 16 most significant bits of the second dword and the 16 least significant bits of the first dword. In case of interrupt no. 0x61, this would be:

```
kd> ? (poi(idtr + (61 * 8) + 4) & 0xffff0000) | (poi(idtr + (61 * 8)) & 0x0000ffff)
Evaluate expression: -2052795048 = 85a4d558
```

As can be seen, the resulting value is greater than the `!idt` output by exactly 0x58 bytes; this is caused by the fact that IDT entries point directly into executable code contained within `KINTERRUPT` instead of the structure base address. As you can imagine, the code starts at offset 0x58 on a Windows 7 32-bit platform (be careful, as the offsets tend to change between system versions, e.g. the code offset is in fact 0x50 in Windows Vista 32-bit):

```
kd> dt _KINTERRUPT
nt!_KINTERRUPT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 InterruptListEntry : _LIST_ENTRY
   +0x00c ServiceRoutine   : Ptr32     unsigned char
   ...
   +0x050 Rsvd1            : Uint8B
   +0x058 DispatchCode     : [135] Uint4B
```

Note that the assembly contained in `DispatchCode` is only a stub which eventually calls the function specified in the `ServiceRoutine` field of the corresponding interrupt descriptor. In order to identify the two interrupts (mouse and keyboard) handled by `i8042prt.sys`, we should therefore do the following:

1. Obtain the kernel base address of `i8042prt.sys` through `NtQuerySystemInformation` with `SystemModuleInformation`.
2. List all `KINTERRUPT` structure pointers found by scanning the IDT.
3. For each interrupt descriptor, subtract the `ServiceRoutine` field value from the driver image base, and keep track of the two interrupts with the smallest deltas.
4. At the end, you will end up with two interrupt indexes which identify the events handled within `i8042prt.sys`.

As we are specifically interested in sniffing key presses and not mouse movements, it is necessary to figure out which interrupt is which. In order to do this, we have to investigate additional fields of the `KINTERRUPT` structure – namely, `Irql` and `SynchronizeIrql`. It turns out that both descriptors share the same synchronization IRQL, yet the `Irql` is equal to `SynchronizeIrql` for keyboard interrupts and is smaller by one for mouse interrupts. This observation alone can be used to decisively determine the purpose of each interrupt vector.

```
kd> dt _KINTERRUPT Irql SynchronizeIrql 85a4d500
nt!_KINTERRUPT
   +0x030 Irql             : 0x5 ''
   +0x031 SynchronizeIrql  : 0x6 ''
kd> dt _KINTERRUPT Irql SynchronizeIrql 85a4d780
nt!_KINTERRUPT
   +0x030 Irql             : 0x6 ''
   +0x031 SynchronizeIrql  : 0x6 ''
```

At this point, a basic understanding of how the `I8042KeyboardInterruptService` function works comes useful. The Windows kernel interrupt handling API guarantees that the second parameter of the function is a pointer to a `DEVICE_OBJECT` structure associated with the physical device (as specified

while registering the handler). Upon entry, the function acquires a pointer to the device extension from `DEVICE_OBJECT.DeviceExtension`:

```
.text:000174C3                mov     eax, [ebp+pDeviceObject]
.text:000174C6                mov     esi, [eax+DEVICE_OBJECT.DeviceExtension]
```

A few instructions later, the driver calls `I8xGetByteAsynchronous` to obtain the code of the pressed / released key, and updates two bytes within the device extension so that they store the two most recently encountered codes:

```
.text:00017581                lea     eax, [ebp+scancode]
.text:00017584                push    eax
.text:00017585                push    1
.text:00017587                call    _I8xGetByteAsynchronous@8
.text:0001758C                lea     eax, [esi+14Ah]
.text:00017592                mov     cl, [eax]
.text:00017594                mov     [esi+14Bh], cl
.text:0001759A                mov     cl, byte ptr [ebp+scancode]
.text:0001759D                mov     [eax], cl
```

Further on, the device driver proceeds to translating the key and inserting it into the system input queue, which is irrelevant for exploitation. It is unclear why `i8042prt.sys` saves the last two scancodes into a private structure; however, this fact is greatly useful for our demonstration purposes. A pointer to the device object can be found in the `KINTERRUPT` structure (which we already located) at offset 0x18, and the device extension can be found similarly at offset 0x28 of the device object. With this information, we can successfuly sniff on all keyboard presses by continuously disclosing the two bytes at offset 0x14a of the extension structure. Additionally, the fact that the last two scancodes are saved also helps, as it makes it possible to synchronize the state better and reduce the volume of missed presses in case the race condition allows for fewer reads per second than the number of presses a user can make.

Keep in mind that there are separate keyboard codes for the "key up" and "key down" events, distinguished by the most significant bit in the byte. Also, in order for the codes to have any meaning to a human, they must be converted using two calls to `MapVirtualKeyEx` with the `MAPVK_VSC_TO_VK` and `MAPVK_VK_TO_CHAR` parameters, respectively. Correct interpretation of key combinations (e.g. capitalizing letters when shift is pressed) would require the exploit to further implement the logic by itself.

A verbose output of a proof of concept sniffer is shown below.

```
[+] i8042prt.sys address: 0x8bd0c000
[+] Candidate: 0x33, service_routine: 0x90b80014, diff: 0x4e74014 irql: 0x90/0x90
[+] Candidate: 0x72, service_routine: 0x8be54df0, diff: 0x148df0 irql: 0x6/0x6
[+] Candidate: 0x91, service_routine: 0x8bd1349a, diff: 0x749a irql: 0x8/0x8
[+] keyboard IDT vector: 0x91
Dumping keyboard input:
time: 1066031, scancode: 28, key: ., pressed: false
time: 1066031, scancode: 156, key: ., pressed: true
time: 1074156, scancode: 18, key: E, pressed: false
time: 1074218, scancode: 45, key: X, pressed: false
time: 1074250, scancode: 146, key: E, pressed: true
time: 1074312, scancode: 173, key: X, pressed: true
time: 1074312, scancode: 25, key: P, pressed: false
time: 1074343, scancode: 38, key: L, pressed: false
time: 1074437, scancode: 153, key: P, pressed: true
time: 1074437, scancode: 166, key: L, pressed: true
time: 1074531, scancode: 24, key: O, pressed: false
time: 1074562, scancode: 152, key: O, pressed: true
time: 1074656, scancode: 23, key: I, pressed: false
time: 1074750, scancode: 151, key: I, pressed: true
time: 1074781, scancode: 20, key: T, pressed: false
time: 1074812, scancode: 148, key: T, pressed: true
```

5.1.5 Patch analysis and affected versions

Race conditions in interacting with user-mode memory are usually trivially addressed by eliminating the root cause of the bug, replacing multiple fetches of a value with copying the data into a local buffer and working on it from there. The `win32k.sys` vulnerabilities discussed in this section were no different: the original `CMP`, `JNB`, `MOV` sequence of instructions was converted to the code presented in Listing 11. In the fixed version of the code, the entire 12-byte long `CALLBACK_OUTPUT` structure is copied to the kernel stack, preventing ring-3 threads from tampering with its contents.

Listing 11: Fixed handling of user-mode callback output data.

```
.text:BF8785FD    mov     esi, [ebp+CallbackOutputPtr]
                  .
                  .
                  .
.text:BF87860B loc_BF87860B:
.text:BF87860B    lea     edi, [ebp+CALLBACK_OUTPUT]
.text:BF87860E    movsd
.text:BF87860F    movsd
.text:BF878610    movsd
                  .
                  .
                  .
.text:BF878639    mov     ecx, [ebp-CALLBACK_OUTPUT.pOutput]
.text:BF87863C    cmp     ecx, eax
.text:BF87863E    jb      short valid_address
; Use ECX as a pointer to a function-specific structure.
```

While the fix introduced in the affected operating systems (all supported Windows-NT family from XP up to 7, both 32- and 64-bit versions) is apparent

and the intention behind it clear, the situation is a little more interesting in Windows 8. The latest operating system has never been subject to the vulnerability, yet analyzing the actual assembly might imply that this was a result of improved compiler optimization rather than deliberate action. The code found in the `win32k.sys` files found in Windows 8 for x86 and ARM are shown in Listings 12 and 13, respectively. As can be seen, the code still uses a user-mode region to fetch a pointer for further use, but only does this once – if the sanitization passes successfully, the old value found in the `ECX` or `R3` register is reused, instead of reading it again. This is a strong indication that the second memory operation was simply optimized out by an improved Windows 8 compiler, accidentally eliminating a security flaw. If this is really the case, the issues make an excellent example of how the robustness of a compiler of choice can introduce or entirely remove a high-impact vulnerability from the resulting binary code.

Listing 12: Secure implementation of pointer sanitization and usage found in Windows 8 x86.

```
.text:BF91800F    mov     edx, [ebp+CallbackOutput]
.text:BF918012    cmp     edx, _W32UserProbeAddress
                  .
                  .
                  .
.text:BF918044    mov     eax, [ebp+CallbackOutput]
.text:BF918047    mov     ecx, [eax+CALLBACK_OUTPUT.pOutput]
.text:BF91804A    cmp     ecx, _W32UserProbeAddress
.text:BF918050    jnb     short invalid_ptr
; Use ECX as a pointer to a function-specific structure.
```

Listing 13: Secure implementation of pointer sanitization and usage found in Windows 8 ARM.

```
.text:000FF2FE    LDR             R4, =W32UserProbeAddress
                  .
                  .
                  .
.text:000FF32C    LDR             R3, [R3,CALLBACK_OUTPUT.pOutput]
.text:000FF32E    LDR             R2, [R4]
.text:000FF330    CMP             R3, R2
.text:000FF332    BCC             valid_address

.text:000FF336 valid_address
; Use R3 as a pointer to a function-specific structure.
```

5.2 CVE-2013-1278

Microsoft explicitly guarantees that each new version of an NT kernel-based Windows would be compatible with prior versions of the system; e.g. all programs developed for Windows XP are supposed to work correctly on Windows 8 without any modifications. This is achieved by preserving compatibility on many fronts, including maintaining support for legacy or deprecated interfaces,

extending existing API functionality instead of completely rebuilding it and so forth. Managing cross-version compatibility is one of the fundamental goals of Windows and therefore is a large and costly effort. More information on application compatibility can be found in a dedicated section in the MSDN library [23].

One of the more important portions of the Windows feature is the *Application Compatibility Database* (SDB in short), a part of the *Shim Engine*. In few words, the database stores information about the compatibility options for each executable file that requires special treatment, such as the symbols and DLL names of files which need to be loaded in older builds, function hooking options etc. Alex Ionescu wrote a brief series of blog posts, providing an in-depth explanation of how the subsystem was designed and implemented [1] [2] [3] [4]. In order to avoid performing costly operations such as hard drive reads each time a legacy application starts, the operating system implements what is called an *Application Compatibility Shim Cache*. In Windows XP/2003, the cache mechanism relied on a section shared across multiple processes; in newer versions of the system, a dedicated `NtApphelpCacheControl` system service was introduced, supposedly in order to further minimize the CPU overhead incurred by each cache-involving operation. Even though the syscall has remained *the* compulsory interface up to the latest Windows 8 system, its internal implementation has gone through significant changes with each major build of the OS. As a consequence, the vulnerability discussed in this section is specific to Windows 7, mitigated by codebase differences in Windows Vista and Windows 8.

5.2.1 The vulnerability

The security flaw itself is trivial in its principle and not related to the functionality it is found in. There are several different operation codes passed to the `NtApphelpCacheControl` function which trigger further, internal handlers:

1. `ApphelpCacheLookupEntry`
2. `ApphelpCacheInsertEntry`
3. `ApphelpCacheRemoveEntry`
4. `ApphelpCacheFlush`
5. `ApphelpCacheDump`
6. `ApphelpCacheSetServiceStatus`
7. `ApphelpCacheForward`
8. `ApphelpCacheQuery`

If you take a deep look into the unpatched versions of `ApphelpCacheLookupEntry` and `ApphelpCacheQuery`, you will notice patterns presented in

Listings 14 and 15. In both assembly snippets, the `EDI` or `EBX` registers store a user-mode pointer. Given what we already know about double fetches, the faulty conditions are apparent – an inconsistency can be caused in between verifying the user-mode pointer and using it as the *dst* parameter of `memcpy`, consequently resulting in a memory write with controlled destination operand, and potentially controlled contents. Such conditions are typically trivial to exploit into a local elevation of privileges, due to the multitude of easily localizable function pointers present in the kernel address space.

Listing 14: The vulnerable portion of the nt!ApphelpCacheLookupEntry implementation.

```
PAGE:00631EC4                   mov     ecx, [edi+18h]
...
PAGE:00631EE0                   push    4
PAGE:00631EE2                   push    eax
PAGE:00631EE3                   push    ecx
PAGE:00631EE4                   call    _ProbeForWrite@12
PAGE:00631EE9                   push    dword ptr [esi+20h]
PAGE:00631EEC                   push    dword ptr [esi+24h]
PAGE:00631EEF                   push    dword ptr [edi+18h]
PAGE:00631EF2                   call    _memcpy
```

Listing 15: The vulnerable portion of the nt!ApphelpCacheQuery implementation.

```
PAGE:007099B5                   mov     edx, [ebx+8Ch]
...
PAGE:007099D7                   push    4
PAGE:007099D9                   push    ecx
PAGE:007099DA                   push    edx
PAGE:007099DB                   call    _ProbeForWrite@12
PAGE:007099E0                   push    [ebp+Length]
PAGE:007099E3                   push    [ebp+P]
PAGE:007099E6                   push    dword ptr [ebx+8Ch]
PAGE:007099EC                   call    _memcpy
```

5.2.2 Winning the race

Unlike the previous bug, this vulnerability allows an attacker to write to an arbitrarily-chosen location in the ring-0 virtual address space. Therefore, it only takes a single race win to escalate one's privileges in the system. Additionally, the available time frame is incomparably longer, as the two memory fetches are separated by dozens of unrelated instructions (primarily the implementation of `ProbeForWrite`). The two facts allow the attacker to not worry about the winning part, and reduce the necessity to use the complicated techniques discussed in prior chapters. It has been shown experimentally that just two threads running on two different CPUs (regardless of Hyper-Threading being enabled or not) successfully trigger the vulnerability in less than 5 seconds in all tested configurations. The exploitability on single-core configurations has

not been tested; yet, it is believed that such platforms should also be prone to exploitation of the issue.

5.2.3 Exploitation

When writing to controlled locations in kernel memory is made possible by a vulnerability, the rest of the process is really just a formality. In order to make an exploit fully reliable and get it to work in every configuration, one should thoroughly understand the contents of memory being copied over to the target address – in this case, the Apphelp Cache structure copied with the faulty `memcpy` call. As this section is for demonstrational purposes only, we will skip this step and go straight into practical exploitation process on a Windows 7 SP1 32-bit platform.

We have empirically demonstrated that trying to execute the `C:\Windows\system32\wuauclt.exe` image always results in having a compatibility entry set up in the cache, making it possible to later use the existing entry while exploiting the `ApphelpCacheLookupEntry` vulnerability, without a need to manually insert a new record with `ApphelpCacheInsertEntry`. Furthermore, in order to reach the vulnerable code path, `NtApphelpCacheControl` requires us to provide a pointer to an empty structure with the following fields set:

1. offset 0x98: A handle to the `C:\Windows\system32\wuauclt.exe` file on disk.
2. offset 0x9c: A `UNICODE_STRING` structure containing an NT path of the file.
3. offset 0xa4: A bogus size of an output buffer, e.g. `0xffffffff`.
4. offset 0xa8: A pointer to a user-mode buffer of a significant size (e.g. 4kB for safety). This is the DWORD being raced against.

Filling in the above data allows an attacker to pass the obligatory call to `ApphelpCacheControlValidateParameters` and results in having the data stored for `wuauclt.exe` in the cache to be copied into the address specified at offset 0xa8 of the input structure (or a completely arbitrary location in case of a successful attack). At this point, we should investigate the *src* and *length* parameters in the critical `memcpy` call in more detail:

```
0: kd> dd esp esp+8
8c60fb4c  0022fcd4 8c974e38 000001c8

0: kd> dd /c 6 poi(esp+4) poi(esp+4)+poi(esp+8)-4
8c974e38  00034782 00000000 00000000 00000000 00000000 00000000
8c974e50  00000000 00000000 00000000 00000000 00000000 00000000
8c974e68  00000000 00000000 00000000 00000000 00000000 00000000
8c974e80  00000000 00000000 00000000 00000000 00000000 00000000
8c974e98  00000000 00000000 00000000 00000000 00000000 00000000
8c974eb0  00000000 00000000 00000000 00000000 00000000 00000000
8c974ec8  00000000 00000000 00000000 00000000 00000000 00000000
8c974ee0  00000001 00000000 00000000 00000000 00000000 00000000
8c974ef8  00000000 00000001 11111111 11111111 11111111 11111111
8c974f10  00000000 00000000 00000000 00000000 00000000 00000000
8c974f28  00000000 00000000 00000000 00000000 00000000 00000000
8c974f40  00000000 00000000 00000000 00000000 00000000 00000000
8c974f58  00000000 00000000 00000000 00000000 00000000 00000000
8c974f70  00000000 00000000 00000000 00000000 00000000 00000000
8c974f88  00000000 00000000 00000000 00000000 00000000 00000000
8c974fa0  00000000 00000000 00000000 00000000 00000000 00000000
8c974fb8  00000000 00000000 00000000 00000000 00000000 00000000
8c974fd0  00000000 00000000 00000000 00000000 00000000 00000000
8c974fe8  00000000 00000000 00000000 00000000 00000000 00000000
```

The overall structure of size 0x1c8 appears to be rather non-interesting, containing zeros, ones and a single 0x00034782 value at the beginning (the specific number differs from system to system, but has always been observed to have 0x0003 in the most significant 16 bits). While this outcome would typically be useful, the size of the structure causes exploitation to be rather troublesome, as overwriting the desired value in the static memory of a device driver or a pool allocation would consequently trash the surrounding area, likely crashing the operating system before we take control. We have encountered a problem of similar nature while trying to take advantage of a NULL pointer dereference vulnerability in the NTFS.SYS driver[7], where it was possible to craft a complex `ERESOURCE` structure at any chosen kernel-mode address. Back then, we decided to employ a new technique – make use of so-called "Private Namespace" objects, originally introduced by Microsoft in Windows Vista. While implemented as regular "namespace" objects, private namespaces have three specific characteristics that make them particularly useful in exploitation scenarios such as this one:

1. They are of variable length, controlled by the creator of the object (the user).

2. They are filled mostly with controlled data, and a great majority of the data found in the object body doesn't matter to the kernel.

3. Each private namespace contains a pointer into a `LIST_ENTRY` structure, used for unlinking when the object is destroyed, effectively allowing for

crafting a four-byte write-what-where condition in operating systems prior to Windows 8 (which has safe unlinking enabled by default).

For details regarding the internal implementation of private namespaces, refer to the aforementioned "Introducing the USB Stick of Death" blog post. In case of the CVE-2013-1278 vulnerability, it is only important that the variable length user-controlled part of their structure is found *after* a pointer to `LIST_ENTRY`. Therefore, we can safely choose to overwrite the pointer with the magic value of 0x00034782 present at the beginning of the *src* buffer, and let the remaining part of the private namespace object "consume" the further 0x1c4 excessive bytes. In order to make the exploit more robust, we can additionally shift the destination address by one byte "right", thus setting the list entry pointer to 0x03?????? (less likely to point into the default process heap), and spray the overall 16-megabyte region with the `LIST_ENTRY` fields (write-what-where condition operands). The states of the object memory prior and following triggering the vulnerability are shown in Figures 14 and 15.

Figure 14: Private namespace object memory prior to winning the race condition.

Once the list entry pointer points into a user-controlled region, we can choose the two values which are going to be written into each other. One commonly used and cross-version compatible vector for kernel write-what-where scenarios is the function pointer found under `nt!HalDispatchTable + 4`, invoked within a nested call in `KeQueryIntervalProfile`; as it is an exported symbol, obtaining its virtual address takes a simple combination of `EnumDeviceDrivers`, `LoadLibraryEx` and `GetProcAddress` calls. Once we properly craft the `Flink` and `Blink` fields of the fake structure and spray it all over the user-mode address space, we can trigger the unlinking by using the `ClosePrivateNamespace` API, which will consequently overwrite the kernel function pointer with the specified value. At this point, we can execute arbi-

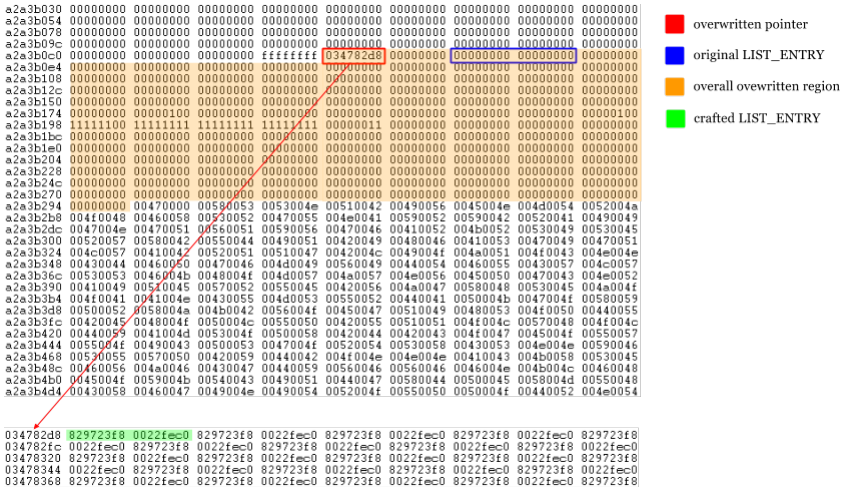

Figure 15: Private namespace object memory after the overwrite takes place.

trary assembly code with CPL=0; the usual direction of choice is to replace the current process' security token with one assigned to the "System" process (pid=4), and later restore the original value of `HalDispatchTable + 4` to maintain system stability. The result of successfully running a proof of concept exploit is illustrated in Figure 16.

One major problem with using private namespaces as "buffers" for taking damage incurred by illegal memory writes of large sizes is that when one of the namespaces adjacent to the damaged one is unlinked or a completely new private namespace is created in the system, the kernel starts traversing the double-linked list and potentially using the bogus (overwritten) contents of the `LIST_ENTRY` structure as memory operands. Where transparency or post-exploitation system reliability is required, it is advisable to manually fix the sensitive portions of the private namespace object (specifically, the list entry).

5.2.4 Patch analysis and variant analysis

Similarly to a majority of double-fetch vulnerabilities, the issue was resolved by simply copying the value in question to kernel address space (a local stack variable optimized to a single register) before reusing it. The fixed implementation of the relevant portion of `ApphelpCacheLookupEntry` is shown in Listing 16. At the time of running Windows against Bochspwn, neither of us had the time to perform a thorough variant analysis in search of similar bugs in the neighbourhood of the affected code – therefore, we originally didn't report an identical bug in `ApphelpCacheQuery`; we only realized that the flaw was there after it was already fixed. This is clear evidence that Microsoft indeed does perform a comprehensive variant analysis over external reports, and also

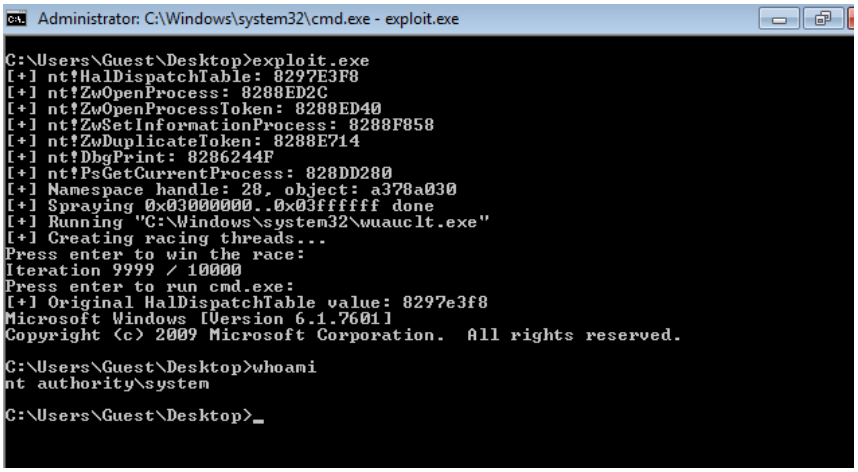

Figure 16: The result of successful CVE-2013-1278 exploitation.

shows that while we only reported so many of Bochspwn's findings, the actual volume of patches applied internally by Microsoft may significantly differ.

Listing 16: A fixed version of nt!ApphelpCacheLookupEntry.

```
PAGE:0063120E                push    4
PAGE:00631210                push    eax
PAGE:00631211                push    ebx
PAGE:00631212                call    _ProbeForWrite@12
PAGE:00631217                push    dword ptr [esi+20h]
PAGE:0063121A                push    dword ptr [esi+24h]
PAGE:0063121D                push    ebx
PAGE:0063121E                call    _memcpy
```

5.3 Double fetch in memcmp

This section describes an interesting quirk in the implementation of one of the memory comparison functions found in Windows kernel mode. It is not resolved at the time of this writing and is rather unlikely to be ever fixed, due to its minimal severity and potential performance hit incurred by a fix. However, we believe the issue still makes an interesting case study, showing how subtle undocumented details of a standard C library function implementation combined with an unaware device driver developer can lead to surprising security loopholes.

While writing a kernel-mode Windows driver in C, a programmer can compare two regions of memory either by calling the standard `memcmp` function, or making use of a Windows-specific `RtlCompareMemory` API, depending on the use-case. The official MSDN documentation does not specify whether the

source parameters of those functions must meet any requirements, besides the following sentence in the *Remarks* section of `RtlCompareMemory`:

> Callers of RtlCompareMemory can be running at any IRQL if both blocks of memory are resident.

The above implies that it is valid to use a user-mode memory region in one of or both parameters, as long as the code is running at a low enough IRQL (preferably `PASSIVE_LEVEL`), e.g. when handling a `METHOD_NEITHER` IOCTL from a user-mode client. Despite the lack of an explicit warning in the documentation, it is otherwise obvious that using a ring-3 pointer instead of operating on a locally-saved copy of the buffer can lead to state inconsistency, if the compared buffer is later used again for a different purpose (resulting in a typical double fetch condition). Furthermore, the degree of control a user-mode application has over its own address space has been shown to make it possible to disclose the contents of a kernel-mode buffer being compared against by taking advantage of *guard pages*[13]. Additionally, timing attacks against memory comparison functions are also feasible and trivial to employ in most scenarios, rendering all security-related usages of memory comparison functions with user-mode parameters (e.g. password verification) insecure. Interestingly, it turns out that there is even more to it – let's start with a brief overview of the functions and how they work.

The semantics of `memcmp` and `RtlCompareMemory` return values are fundamentally different – while the former informs the caller about the relationship between the contents of the memory blocks, the latter simply returns the length of the matching prefix. This causes both functions to have separate implementations, which additionally differ between versions of Windows and – obviously – different bitnesses. Below are a few examples of algorithms implemented by the functions on various platforms.

In Windows 7/8 32-bit, `RtlCompareMemory` works as follows:

1. Compare 32-bit chunks of memory for as long as possible.
2. If there is a 32-bit mismatch, compare the double words in byte granularity, return an internal counter when the offending offset is found.
3. Otherwise, compare the remaining $0 - 3$ bytes at the end of the regions, breaking on mismatch.
4. Return the internal counter.

In Windows 7/8 64-bit, `memcmp` works as follows:

1. Compare 64-bit chunks of memory in an unfolded loop consisting of four comparisons (i.e. 32 bytes are compared in one iteration).
2. If there is a 64-bit mismatch, swap both quad words using `BSWAP` and return the result of the $-(x \leq y)$ expression.

3. Compare the remaining 0 – 7 bytes.

4. If there is a byte mismatch, return $-(x \leq y)$.

5. Return 0.

Both of the above algorithms contain a potential double fetch – when two four or eight-byte values differ, the functions *come back* and fetch the bytes to perform a comparison with a higher resolution. A typical goal in attacking those implementations would be to fake matching regions, i.e. get `memcpy` to return 0 or `RtlCompareMemory` to return *Length* when the buffers are not really identical. If you take a closer look, it turns out that neither implementation can be abused in such a way. In case of the first one, the routine by design cannot return a value larger than the number of matched bytes, so exploiting the double fetch doesn't gain the attacker any benefit. For the second one, the only two possible return values after a mismatch are -1 and 1 due to the formula used.

The situation is dramatically different in case of the `memcmp` implementation found in Windows 8 32-bit at the time of this writing, as presented in Listing 17 in the form of a C-like pseudocode. The problem with the function is that it assumes that once a four-byte mismatch occurs, there must be at least a single pair of different bytes during the second run. If this is not the case due to data inconsistency caused by a rogue user and all four bytes match, the routine returns 0. In other words, this quirk makes it possible to reduce the memory comparison process from full n to just four bytes, by flipping the first dword in the compared buffer so that it is different from the dword in the other region when accessed the first time, but identical in the second run. If the expected 32-bit value at offset 0 is known (for example, it is a magic number), the race condition can be exploited to convince the `memcmp` caller that the input buffer is equivalent to whatever it was compared against, while really only matching the first 32 bits. Otherwise, the double word can be brute-forced due to the relatively small 0 to $2^{32} - 1$ range.

Listing 17: Pseudocode of the Windows 8 32-bit kernel "memcmp" function.

```
int memcmp(const void *ptr1, const void *ptr2, size_t num) {
  while (num >= sizeof(DWORD)) {
    if (*(PDWORD)ptr1 != *(PDWORD)ptr2) {
      num = sizeof(DWORD);
      break;
    }
    ptr1 += sizeof(DWORD);
    ptr2 += sizeof(DWORD);
    num -= sizeof(DWORD);
  }

  while (num > 0) {
    BYTE x = *(PBYTE)ptr1;
    BYTE y = *(PBYTE)ptr2;
    if (x < y) {
      return -1;
    } else if (y > x) {
      return 1;
    }
    ptr1++; ptr2++;
    num--;
  }

  return 0;
}
```

The behavior is difficult to clearly classify as a "bug", as it already requires a somewhat erroneous condition to occur. It is also only specific to a single Windows version and bitness, making it something more of a curiosity than an actual problem. Microsoft was informed about the finding and eventually decided not to fix it. If nothing more, it is a solid argument for copying user-provided data into local buffers prior to performing any operations against it, even if the operation in question is just a trivial C library function call.

6 Future work

There are multiple areas for development at each level of work presented in this paper. First of all, the current state of the Bochspwn project based on the *Bochs* emulator instrumentation framework has many shortcomings – first and foremost, due to the nature of running a full-fledged operating system on a software emulator with run-time execution analysis, it is painfully slow, nearly on the verge of losing interaction with the guest system. However there are still parts of the current implementation that could be further optimized for CPU usage, the low performance of Bochs itself is the primary restrictive factor which cannot be mitigated in any way other than changing the fundamental design of the project. One idea for a new, massively more efficient design is to move away from software emulation of the x86(-64) architecture and instead instrument the operating system by taking advantage of the virtualization technology imple-

mented by Intel and AMD. The new concept assumes the usage of a thin VMM similar to the one used in the *Blue Pill* proof of concept developed by Joanna Rutkowska[9], instrumenting kernel-to-user memory references of a running operating system on the fly. The overhead incurred by the hypervisor would be caused exclusively by the handling of the special type of memory operations, which are typically less than 0.01% of the total OS run time; we believe it to be sufficiently small for the instrumentation to be transparently used on physical machines used for one's daily work. This project, codenamed *HyperPwn* to distinguish the technology used, is currently in the works and is planned to be presented at the BlackHat USA 2013 Briefings security conference.

With regards to Bochspwn itself, the project is yet to be tested against the Linux, Unix and BSD kernels, an effort that is also being worked on at the time of writing this paper. Concluding by the success Bochspwn had with Microsoft Windows and depending on the results with other platforms, it might be beneficial for local security of operating system kernels to further investigate the capabilities of CPU-level execution environment instrumentation. We strongly believe that the general concept has an enormous potential which largely exceeds just finding local *time of check to time of use* vulnerabilities, as there are many further code or memory access patterns that could be used to detect otherwise overlooked instances of unusual system behavior, potentially putting its security at risk. Several ideas of potentially interesting patterns can be found below:

- Failed attempts to access non-existing memory user or kernel-mode memory pages, possibly absorbed by try/except constructs and therefore not manifested in the form of a system crash.
- Multiple writes to the same user-mode memory area in the scope of a single system service (effectively the opposite of current Bochspwn implementation), with the potential of the finding instances of accidentally disclosed sensitive data (e.g. uninitialized pool bytes) for a short while, before being replaced with the actual syscall output.
- Execution of code with CPL=0 from user-mode virtual address space (a condition otherwise detected by the *Supervisor Mode Execution Protection* mechanism introduced in latest Intel processors) or execution of code from non-executable memory regions which are not subject to *Data Execution Prevention*, such as non-paged pool in Windows 7[24].
- Accessing pageable (yet not swapped to disk) memory while at the `DISPATCH_LEVEL` or higher IRQL.

These and many other patterns could be used in stand-alone manner, or as a complementary part of kernel-mode fuzzing (e.g. system call or IOCTL fuzzing) performed within the guest. With the current implementation of Bochs, it is essential that the patterns are conceptually simple and only require the instrumentation to know as little about the internal system state as possible –

otherwise, both CPU and memory overheads start growing beyond acceptable extent. This might possibly change with *HyperPwn*, yet it will always hold true that the more generic and system-independent pattern, the more effectively the instrumentation can perform. Once both projects are open-sourced, we highly encourage you to experiment with the above, or your own pattern ideas.

Despite the dynamic approach that we originally decided to use to tackle the problem, it is likely that static analysis could be employed to identify *double fetch* memory problems similarly, or even more effectively than Bochspwn. One of the fundamental problems with the detection of erroneous behavior at system run-time is that it is limited to the code coverage achieved during the testing process, which is often very small in comparison to the overall code base of a product. While the problem can be partially solved for applications parsing complex, self-contained file formats by gathering a large corpus of input files and distilling it to a compact-sized set of files with a maximum coverage, the problem is significantly more difficult with regards to testing operating system kernels. Instead of using a single binary blob describing the program functionality required or triggered by a particular testcase, obtaining kernel code coverage requires one to understand and implement the logic of each system service or a group of those, a task which cannot be easily automated. The benefit to using static analysis would therefore be remarkable, since once implemented, the algorithm could be applied to all components of the kernel address space with relative ease (e.g. defining kernel structures and input data entrypoints for syscalls), including those unlikely to ever be tested by a coverage-based analysis tool. We expect static analysis to be the future of the vulnerability hunting industry, and race conditions in interfacing with user-mode memory seems to be a great first target, judging by the very basic concept behind it (using the same memory address twice within a specific code execution path).

Last but not least, very little research in the past has focused on practical exploitation of race condition vulnerabilities in the Windows environment, be it ring-0 or ring-3 problems. This obviously correlates with the volume of issues of this type identified in the operating system, in comparison to other, more prevalent problems such as regular memory corruptions. This paper aims to show that contrary to popular belief, there indeed is a number of exploitable race conditions in Windows and other software platforms – they just take a different methodology to find. While several techniques were presented by researchers in the past and in previous sections of the document, this area still needs further work, starting from an in-depth analysis of the system scheduler in the exploitation context, through techniques specific to certain configurations (e.g. single CPU or NUMA-enabled machines), up to new means of extending attack time windows by taking advantage of the specific implementations and quirks found in both operating systems and chips alike.

In general, we hope that this work will inspire more development in the fields of detecting and exploiting time-bound vulnerabilities, as well as extending the usage of CPU-level instrumentation to identify other types of problems in operating systems.

7 Remarks

As we have shown in the paper, not all vulnerability classes in the Windows kernel have been audited for and mitigated equally well. It is difficult to pinpoint a single most important reason for why the *time of check to time of use* bugs covered in this document exist in the first place – perhaps one of the main factors is the current model of passing input data for processing to kernel-mode handlers, contained within a number of multi-layered structures referencing each other through user-mode pointers. While most threats related to processing input data are repulsed with the usage of the *previous mode* and fetching of all input data into temporary buffers in the top-level syscall handlers, it is nevertheless not uncommon to see a kernel routine executing within a callstack a few levels deep to operate on a user-mode pointer. There is no easy resolution for the situation – in order to improve the current posture, each system call would require to be carefully reviewed and possibly restructured, a task which is both dubious and – considering the still very low relation of vulnerable to all existing code paths – not cost efficient. Therefore, we believe that the most viable way of ensuring that the kernel is free from the discussed type of race conditions is to invest resources in building effective tools to find them, such as Bochspwn. In fact, it is rather surprising that neither Microsoft nor security researchers around the world seem to have attempted to approach the problem in a dedicated way before.

Local privilege escalation vulnerabilities are usually of little interest to black-hat hackers or exploit brokers' customers, which explains the little work done in the kernel security area during the last decade. As sandboxing technologies are now commonly used in nearly all popular web browsers and other widely-deployed client software (e.g. Adobe Reader), most kernel vulnerabilities can be successfully chained together with client exploits, effectively allowing for a full system compromise; most vulnerabilities discovered by Bochspwn could be currently used as a sandbox break-out in a number of currently available sandboxing implementations. However, the exploitation of all of the issues could be partially mitigated by limiting the affinity mask of each renderer process to a single CPU, or fully mitigated by disallowing the usage of more than one thread per renderer. Additionally, enabling `ProcessSystemCallDisablePolicy` in Windows 8 could also prevent the exploitation of all security issues found in the user/gdi kernel subsystem.

For something completely different, it is interesting to take a perspective look at the recent evolution of user and kernel-mode vulnerability hunting process. The professional part of the information security industry working on identification and exploitation of client software flaws has generally stabilized around bugs in popular web browsers and their plugins, mostly focusing on just two types of violations – use-after-free and memory corruption (buffer overflows), with few examples of logical, design or any other issues used in practical attacks. On the other hand, every now and then the Windows kernel surprises security professionals with a completely new or known but largely ignored vector of attacks, which all of the sudden leads to identifying dozens and dozens of

new local privilege escalation vulnerabilities, one example being Tarjei Mandt's "Kernel Attacks through User-Mode Callbacks" research in 2011[33], which led to more than 40 unique fixes in the `win32k.sys` device driver, all of them having a common root cause. This very research and its results might indicate that there may in fact be more undiscovered types of problems specific to kernel-mode or even just particular device drivers, manifested in the form of a few dozens of critical vulnerabilities. Let's wait and see.

8 Conclusion

In this paper, we have discussed how specific patterns of access to user-mode memory can introduce major security issues in the Windows kernel, and how the authors' exemplary implementation of CPU-level operating system instrumentation can help effectively identify instances of such race conditions. Further in the document, we have addressed a number of techniques to facilitate the exploitation of *time of check to time of use* vulnerabilities in the kernel, and later shown how they could successfully become a part of a practical attack, on the example of several different findings of the Bochspwn project, including leveraging a non-memory corruption issue to compromise the system security. In an attempt to motivate more development in the field of race condition hunting and exploitation, we have listed several areas which still require further research, and conclusively provided some final remarks.

9 Updates

1. 05/06/2013: Updated references, updated Figure 5, inserted Listing 8, introduced minor changes in the *Page boundaries* section. We would like to thank Alexander Peslyak for pointing out several problems in the paper.

References

[1] Alex Ionescu: *Secrets of the Application Compatilibity Database (SDB) Part 1*. `http://www.alex-ionescu.com/?p=39`.

[2] Alex Ionescu: *Secrets of the Application Compatilibity Database (SDB) Part 2*. `http://www.alex-ionescu.com/?p=40`.

[3] Alex Ionescu: *Secrets of the Application Compatilibity Database (SDB) Part 3*. `http://www.alex-ionescu.com/?p=41`.

[4] Alex Ionescu: *Secrets of the Application Compatilibity Database (SDB) Part 4*. `http://www.alex-ionescu.com/?p=43`.

[5] Alexander Viro: *CAN-2005-2490 sendmsg compat stack overflow*. `https://bugzilla.redhat.com/show_bug.cgi?id=166248`.

[6] Fermin J. Serna: *MS08-061 : The case of the kernel mode double-fetch.* http://blogs.technet.com/b/srd/archive/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch.aspx.

[7] Gynvael Coldwind, Mateusz "j00ru" Jurczyk: *Introducing the USB Stick of Death.* http://j00ru.vexillium.org/?p=1272.

[8] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, Edward W. Felten: *Lest We Remember: Cold Boot Attacks on Encryption Keys.* http://static.usenix.org/event/sec08/tech/full_papers/halderman/halderman.pdf.

[9] Joanna Rutkowska: *Subverting Vista Kernel For Fun And Profit.* http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf.

[10] Jonathan Morrison: *http://blogs.msdn.com/b/itgoestoeleven/archive/2008/03/31/why-your-user-mode-pointer-captures-are-probably-broken.aspx.* http://blogs.msdn.com/b/itgoestoeleven/archive/2008/03/31/why-your-user-mode-pointer-captures-are-probably-broken.aspx.

[11] Ken Johnson, Matt Miller: *Exploit Mitigation Improvements in Windows 8.* http://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf.

[12] Mark Russinovich, David A. Salomon, Alex Ionescu: *Windows Internals 6, Part 1.* Intel Corporation, 2012.

[13] Mateusz "j00ru" Jurczyk: *Fun facts: Windows kernel and guard pages.* http://j00ru.vexillium.org/?p=1594.

[14] Mateusz "j00ru" Jurczyk: *The story of CVE-2011-2018 exploitation.* http://j00ru.vexillium.org/blog/20_05_12/cve_2011_2018.pdf.

[15] Mateusz "j00ru" Jurczyk: *Windows Security Hardening Through Kernel Address Protection.* http://j00ru.vexillium.org/?p=1038.

[16] Mateusz "j00ru" Jurczyk: *Windows X86-64 System Call Table (NT/2000/XP/2003/Vista/2008/7/8).* http://j00ru.vexillium.org/ntapi_64/.

[17] Mateusz "j00ru" Jurczyk: *Windows X86 System Call Table (NT/2000/XP/2003/Vista/2008/7/8).* http://j00ru.vexillium.org/ntapi/.

[18] Mateusz "j00ru" Jurczyk, Gynvael Coldwind: *Exploiting the otherwise non-exploitable: Windows Kernel-mode GS Cookies subverted.* http://vexillium.org/dl.php?/Windows_Kernel-mode_GS_Cookies_subverted.pdf.

[19] Microsoft: *Microsoft Security Bulletin MS13-016 - Important.* http://technet.microsoft.com/en-us/security/bulletin/MS13-016.

[20] Microsoft: *Microsoft Security Bulletin MS13-017 - Important.* http://technet.microsoft.com/en-us/security/bulletin/MS13-017.

[21] Microsoft: *Microsoft Security Bulletin MS13-031 - Important.* http://technet.microsoft.com/en-us/security/bulletin/MS13-031.

[22] Microsoft: *Microsoft Security Bulletin MS13-036 - Important.* http://technet.microsoft.com/en-us/security/bulletin/MS13-036.

[23] MSDN: *Application Compatibility.* http://technet.microsoft.com/en-us/library/ee461265(v=ws.10).aspx.

[24] MSDN: *Data Execution Prevention.* http://technet.microsoft.com/en-us/library/cc738483%28v=ws.10%29.aspx.

[25] MSDN: *Scheduling.* http://msdn.microsoft.com/en-us/library/windows/desktop/ms685096%28v=vs.85%29.aspx.

[26] MSDN: *Working Set.* http://msdn.microsoft.com/en-us/library/windows/desktop/cc441804(v=vs.85).aspx.

[27] mxatone: *Analyzing local privilege escalations in win32k.* http://uninformed.org/index.cgi?v=10&a=2.

[28] Niels Provos: *Improving Host Security with System Call Policies.* http://www.citi.umich.edu/u/provos/papers/systrace.pdf.

[29] Ralf Hund, Carsten Willems, Thorsten Holz: *Practical Timing Side Channel Attacks Against Kernel Space ASLR.* http://www.internetsociety.org/doc/practical-timing-side-channel-attacks-against-kernel-space-aslr.

[30] Robert N. M. Watson: *Exploiting Concurrency Vulnerabilities in System Call Wrappers.* http://www.watson.org/~robert/2007woot/2007usenixwoot-exploitingconcurrency.pdf.

[31] sgrakkyu, twiz: *Attacking the Core : Kernel Exploiting Notes.* http://phrack.org/issues.html?issue=64&id=6#article.

[32] Tal Garfnkel: *Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools*. http://www.cs.berkeley.edu/~dawnsong/teaching/f12-cs161/readings/traps.pdf.

[33] Tarjei Mandt: *Kernel Attacks through User-Mode Callbacks*. http://www.mista.nu/research/mandt-win32k-paper.pdf.

[34] The WINE Project: *Wine Conformance Tests*. http://wiki.winehq.org/ConformanceTests.

[35] twiz, sgrakkyu: *From RING 0 to UID 0*. http://events.ccc.de/congress/2007/Fahrplan/attachments/1024_Ring-Zero-to-UID-Zero.