Detecting Kernel Memory Disclosure with x86 Emulation and Taint Tracking

Mateusz Jurczyk
mjurczyk@google.com

Google LLC

June 2018

Abstract

One of the responsibilities of modern operating systems is to enforce privilege separation between user-mode applications and the kernel. This includes ensuring that the influence of each program on the execution environment is limited by the defined security policy, but also that programs may only access the information they are authorized to read. The latter goal is especially difficult to achieve considering that the properties of C – the main programming language used in kernel development – make it highly challenging to securely pass data between different security domains. There is a significant risk of disclosing sensitive leftover kernel data hidden amidst the output of otherwise harmless system calls, unless special care is taken to prevent the problem. Issues of this kind can help bypass security mitigations such as KASLR and StackGuard, or retrieve information processed by the kernel on behalf of the system or other users, e.g. file contents, network traffic, cryptographic keys and so on.

In this paper, we introduce the concept of employing full system emulation and taint tracking to detect the disclosure of uninitialized kernel stack and heap/pool memory to user-space, and discuss how it was successfully implemented in the *Bochspwn Reloaded* project based on the open-source Bochs IA-32 emulator. To date, the tool has been used to identify over 70 memory disclosure vulnerabilities in the Windows kernel, and more than 10 lesser bugs in Linux. Further in the document, we evaluate alternative ways of detecting such information leaks, and outline data sinks other than user-space where uninitialized memory may also leak from the kernel. Finally, we provide suggestions on related research areas that haven't been fully explored yet. Appendix A details several further ideas for system-wide instrumentation (implemented using Bochs or otherwise), which can be used to discover other programming errors in OS kernels.

Contents

1 Introduction

Modern operating systems running on x86/x86-64 CPUs are multi-threaded and use a client—server model, where user-mode applications (*clients*) execute independently of each other, and only call into the kernel (*server*) once they intend to operate on a resource managed by the system. The mechanism used by code running in ring 3 to call into a predefined set of ring 0 functions is known as *system calls* or *syscalls* in short. While CPU registers are used to return the exit code and to pass the first few arguments on 64-bit platforms, the primary I/O data exchange channel is user-mode memory. Consequently, the kernel continuously operates on ring 3 memory during its activity. The life of a typical system call is illustrated in Figure 1.

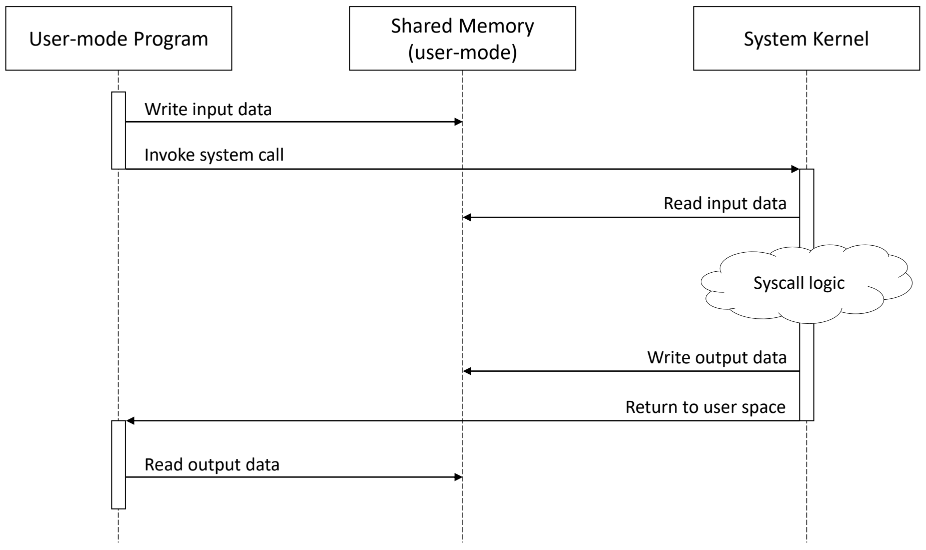

Figure 1: Life of a system call

The execution flow is simple in principle, but the fact that user-mode memory is a shared resource that can be read or written to at any point in time by another thread makes the kernel's interactions with the memory prone to race conditions and other errors, if not implemented carefully. One way to achieve a secure implementation is to have every system call handler adhere to the following two rules (in that order):

1. Fetch data from each user-mode memory location at most once, with an active exception handler in place, and save a local copy in the kernel space for further processing.

2. Write to each user-mode memory location at most once, with an active exception handler in place, only with data intended for the user-mode client.

The usage of pointer annotations (such as `__user` in Linux) and dedicated functions for operating on client memory (e.g. `copy_from_user`, `copy_to_user`, `get_user`, `put_user` in Linux) helps maintain a healthy user-mode interaction hygiene, by forcing developers to make conscious decisions about when to perform these operations. On the other hand, the looser approach observed in the Windows kernel (direct pointer manipulation) seems to provoke the presence of more security issues, as demonstrated later in the paper.

Breaking each of the above requirements has its own unique consequences, with varying degree of impact on system security:

- **Double fetches** – caused by reading a user-mode value more than once within the same semantic context, and expecting it to remain consistent between the reads. This introduces *time of check to time of use* race conditions which may lead to buffer overflows, write-what-where conditions and other memory safety violations, depending on the affected code. The work discussed in this paper is largely based on the original Bochspwn project, developed in 2013 to detect double fetch vulnerabilities [68, 69, 70].

- **Read-after-write conditions** – caused by writing to a user-mode memory address and subsequently reading from it with the assumption that the written value has remained unchanged. These race conditions have security impact similar to that of double fetches.

- **Double writes** – caused by writing sensitive information to user-mode memory, and later overwriting it with legitimate data in the scope of the same system call. They result in information disclosure of limited duration, most commonly involving kernel-space addresses.

- **Unprotected memory operations** – caused by the lack of exception handlers set up around user-mode memory accesses. They may be exploited to trigger unhandled kernel exceptions and consequently crash the operating system.

The above bug classes are addressed in more detail in Appendix A. The primary subject of this paper is the breaking of the following rule:

> *Write to each user-mode memory location at most once, [...],* ***only with data intended for the user-mode client****.*

Due to a combination of circumstances such as the nature of the C programming language, current state of compilers, design of system allocators and common code optimization techniques, it is difficult to avoid unintentionally leaking kernel memory while interacting with user-mode programs. In the next section, we discuss the origins of the problem, its impact on system security and prior work done to address it.

2 Memory disclosure in operating systems

A disclosure of privileged system memory occurs when the kernel returns a larger region of data than is necessary to store the relevant information contained within. Frequently, the redundant bytes originate from a kernel memory region which used to store data previously processed in another context, and are not pre-initialized to ensure that the old values are not propagated to new data structures.

In some cases, fault can be clearly attributed to insufficient initialization of certain variables or structure fields in the code. At other times, information leaks occur in the kernel even though the corresponding source code is seemingly correct (sometimes only on specific CPU architectures). In either case, memory disclosure between different privilege levels running C code is a problem hardly visible to the naked eye.

2.1 C language-specific properties

In this section, we outline several aspects of the C language that are most relevant to the problem in question.

2.1.1 Indeterminate state of uninitialized variables

Standalone variables of simple types such as `char` or `int`, as well as members of larger data structures (arrays, structures and unions) remain in an *indeterminate* state until first initialized, if they are stored on the stack (formally under *automatic storage duration*) or heap (*allocated storage duration*). The relevant citations from the C11 N1570 specification draft [14] are as follows (emphasis added by author):

> 6.7.9 Initialization
> ...
> 10 If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.
>
> 7.22.3.4 The malloc function
> ...
> 2 The malloc function allocates space for an object whose size is specified by size and whose value is indeterminate.
>
> 7.22.3.5 The realloc function
> ...
> 2 The realloc function deallocates the old object pointed to by ptr and returns a pointer to a new object that has the size specified by size. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

The part most applicable to system code is the one concerning stack-allocated objects, as kernels typically have dynamic allocation interfaces with their own semantics (not necessarily consistent with the C standard library, as discussed in Section 2.2.1 “Memory reuse in dynamic allocators”).

To our best knowledge, none of the three most popular C compilers for Windows and Linux (Microsoft C/C++ Compiler, gcc, LLVM) produce code which pre-initializes otherwise uninitialized objects on the stack in *release mode* (or equivalent). There are compiler switches enabling the poisoning of stack frames with marker bytes (e.g. `/RTCs` in Microsoft Visual Studio), but they are not used in production builds for performance reasons. As a result, uninitialized objects on the stack inherit old values of their corresponding memory areas.

Let’s consider an example of a fictional Windows system call implementation, which multiplies the input integer by two and returns the product (Listing 1). It is evident that in the corner case of `InputValue=0`, the `OutputValue` variable remains uninitialized and is copied in that form back to the client. Such a bug would enable the disclosure of four bytes of kernel stack memory on each syscall invocation.

```
NTSTATUS NTAPI NtMultiplyByTwo(DWORD InputValue, LPDWORD OutputPointer) {
  DWORD OutputValue;

  if (InputValue != 0) {
    OutputValue = InputValue * 2;
  }

  *OutputPointer = OutputValue;
  return STATUS_SUCCESS;
}
```

Listing 1: Memory disclosure via a local uninitialized variable

While possible, information leaks through standalone variables are not very common in practice, as modern compilers will often detect and warn about such problems, and being functional bugs, they may also be identified during development or testing. However, a second example in Listing 2 shows that a leak may also take place through a structure field.

In this case, the `Reserved` field is never explicitly used in the code, but is still copied back to user-mode, and thus also discloses four bytes of kernel memory to the caller. This example distinctly shows that having every field of every structure returned to the client initialized on every code path is a difficult task, and in many cases it is plainly counterintuitive to do so, if the field in question does not play any practical role in the code.

Overall, the fact that uninitialized variables on the stack and in dynamic allocations take the contents of data previously stored in their memory regions lies at the core of the kernel memory disclosure problem.

```
typedef struct _SYSCALL_OUTPUT {
  DWORD Sum;
  DWORD Product;
  DWORD Reserved;
} SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;

NTSTATUS NTAPI NtArithOperations(
  DWORD InputValue,
  PSYSCALL_OUTPUT OutputPointer
) {
  SYSCALL_OUTPUT OutputStruct;

  OutputStruct.Sum = InputValue + 2;
  OutputStruct.Product = InputValue * 2;

  RtlCopyMemory(OutputPointer, &OutputStruct, sizeof(SYSCALL_OUTPUT));

  return STATUS_SUCCESS;
}
```

Listing 2: Memory disclosure via a reserved structure field

2.1.2 Structure alignment and padding bytes

The initialization of all members of an output structure is a good start to avoid memory disclosure, but it is not always sufficient to guarantee that uninitialized bytes will not appear in its low-level representation. Let's consider the following excerpts from the C11 specification draft:

> 6.5.3.4 The sizeof and _Alignof operators
> ...
> 4 [...] When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding.
>
> 6.2.8 Alignment of objects
> 1 Complete object types have alignment requirements which place restrictions on the addresses at which objects of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. [...]
>
> 6.7.2.1 Structure and union specifiers
> ...
> 17 There may be unnamed padding at the end of a structure or union.

Practically speaking, C compilers for the x86(-64) architectures apply *natural alignment* to structure fields of primitive types, which means that each such field is N-byte aligned, where N is the given field's width. Furthermore, entire structures and unions are also aligned such that when they are declared in an array, the alignment requirements of the nested fields are still met. In order to accommodate the alignment, padding bytes are artificially inserted into structures where necessary[1]. While not directly accessible in the source code, these bytes also inherit old values from the underlying memory regions and may leak information to user-mode, if they are not reset in time.

In the example shown in Listing 3, a `SYSCALL_OUTPUT` structure is passed back to the caller. It contains a 4-byte and an 8-byte field, separated by 4 bytes of padding required to align `LargeSum` to an 8-byte boundary. Even though both fields are properly initialized, the padding bytes are not explicitly set, which again leads to a kernel stack memory disclosure. The specific layout of the structure in memory is illustrated in Figure 2.

```
typedef struct _SYSCALL_OUTPUT {
  DWORD Sum;
  QWORD LargeSum;
} SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;

NTSTATUS NTAPI NtSmallSum(
  DWORD InputValue,
  PSYSCALL_OUTPUT OutputPointer
) {
  SYSCALL_OUTPUT OutputStruct;

  OutputStruct.Sum = InputValue + 2;
  OutputStruct.LargeSum = 0;

  RtlCopyMemory(OutputPointer, &OutputStruct, sizeof(SYSCALL_OUTPUT));

  return STATUS_SUCCESS;
}
```

Listing 3: Memory disclosure via structure padding

Leaks through alignment holes are relatively common, as a variety of complex structures are used in user↔kernel communication in modern systems. This is especially true for 64-bit platforms, where the width of pointers, `size_t` and similar types increases from 4 to 8 bytes, thus also increasing the number of padding bytes necessary to align such structure fields.

As padding bytes cannot be directly addressed in the source code, it is intuitive to use the `memset` or analogous function to reset the entire memory

[1]Typically, a maximum of 7 padding bytes can be observed between two consecutive fields, if they are of type `char` and `long long` (or equivalent), respectively. In some rare and extreme cases, the padding can be as long as 15, 31 or even 63 bytes, for such esoteric types as 80-bit `long double`, `__m256` or `__m512`.

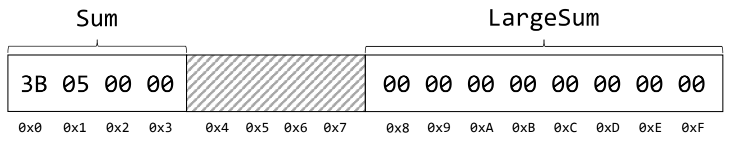

Figure 2: Layout of a structure with padding bytes

region prior to initializing any of its fields and copying it to user-mode, like so:

```
memset(&OutputStruct, 0, sizeof(OutputStruct));
```

However, Seacord R. C. argues [100] that it is a noncompliant solution, because the padding bytes may still be clobbered after the `memset` call, e.g. as a side effect of operating on the adjacent fields. The concern may be justified by the following statement in the C specification draft:

> 6.2.6 Representations of types
> 6.2.6.1 General
> ...
> 6 When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values. [...]

In practice, none of the C compilers we have tested read or write beyond the memory regions of the explicitly declared fields. This sentiment also seems to be shared by operating system vendors and kernel developers, who use `memset` as their preferred fix of choice.

The aforementioned publication lists the following *compliant* ways of addressing the problem:

- Serializing the structure contents into a continuous, dense buffer of bytes, and unserializing it back on the other side.
- Explicitly declaring the padding areas as dummy structure fields, and manually zero-initializing them in the code.
- Enabling structure packing to completely disable memory alignment, thus eliminating the problematic padding bytes.

Evaluating the pros and cons of each of the above solutions is left as an exercise to the reader.

2.1.3 Unions and diversely sized fields

Unions are another intricate construct in the context of cross-privilege level communication in the C language. The relevant parts of the C11 specification draft explaining how unions are represented in memory are quoted on the next page.

6.2.5 Types

. . .

20 Any number of derived types can be constructed from the object and function types, as follows: [...] A union type describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

6.7.2.1 Structure and union specifiers

. . .

6 As discussed in 6.2.5, a structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of members whose storage overlap.

. . .

16 The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time.

The problem here is that if a union consists of several fields of various widths, and only one of the smaller fields is set, then the remaining bytes allocated to accommodate the larger ones are left uninitialized. Let's examine an example of an imaginary system call handler presented in Listing 4, together with the memory layout of the `SYSCALL_OUTPUT` union illustrated in Figure 3.

```
typedef union _SYSCALL_OUTPUT {
  DWORD Sum;
  QWORD LargeSum;
} SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;

NTSTATUS NTAPI NtSmallSum(
  DWORD InputValue,
  PSYSCALL_OUTPUT OutputPointer
) {
  SYSCALL_OUTPUT OutputUnion;

  OutputUnion.Sum = InputValue + 2;

  RtlCopyMemory(OutputPointer, &OutputUnion, sizeof(SYSCALL_OUTPUT));

  return STATUS_SUCCESS;
}
```

Listing 4: Memory disclosure via a partially initialized union

As can be seen, the total size of the `SYSCALL_OUTPUT` union is 8 bytes, due to the width of the larger `LargeSum` field. However, the function only sets the value of the smaller field, leaving the 4 trailing bytes uninitialized and subsequently disclosed to the client application.

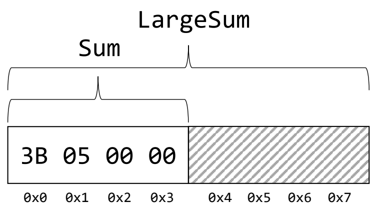

Figure 3: Memory layout of a partially initialized union

A safe implementation should only set the `Sum` field in the user address space, instead of copying the entire object with potentially unused regions of memory. Another frequently seen fix is the usage of the `memset` function to reset the kernel's copy of the union prior to setting any of its fields and passing it back to user-mode.

2.1.4 `sizeof` considered harmful

As shown in the two previous subsections, the usage of the `sizeof` operator may (in)directly contribute to the disclosure of kernel memory, by provoking the copying of more data than the kernel had previously initialized. This manifests a broader problem of having to weight code cleanliness and brevity against system security, often with the kernel developer only having a blurry understanding of the outcome of each possible decision.

To put it in other words, the C language lacks the apparatus necessary to securely move data from the kernel to userland – or more generally, between any differently privileged security contexts. It provides no runtime metadata that can help establish which bytes have been set in each data structure used for user$\leftrightarrow$kernel communication. As a result, the burden is on the programmer to determine which portions of each object contain relevant data and should be passed to the caller. If done properly, this would require introducing a dedicated *safe-copy* function for every output data structure used by the kernel, which in turn would bloat the code size, likely degrade its readability, and overall be a tedious and time-consuming task.

On the other hand, it is convenient to avoid the hassle and simply copy an entire kernel memory region with a single `memcpy` call and a `sizeof` argument, and let the client determine which parts of the output buffer it will actually use. This seems to be the state of Windows and Linux today. When a particular instance of an information leak is reported, a patch with a `memset` call is promptly submitted and distributed, but the deeper problem remains.

This is why we believe that even with tools for automated detection of kernel memory disclosure (such as Bochspwn Reloaded), the bug class may persist for many years, if systemic solutions are not implemented in the language specification, compilers and operating systems to address the problem on a more general level.

2.2 System-specific properties

There are certain kernel design decisions, programming practices and code patterns which affect how prone operating systems are to memory disclosure vulnerabilities. They are briefly discussed in the following subsections.

2.2.1 Memory reuse in dynamic allocators

Current dynamic memory allocators (in both user and kernel-mode) are highly optimized, as their performance has a considerable impact on the performance of the entire operating system. One of the most important optimizations is *memory reuse* – when a heap/pool allocation is freed, the corresponding memory is rarely completely discarded, but is instead saved in a list of regions ready to be returned when the next allocation request is received. To save CPU cycles, the memory regions are not cleared by default between being used by two different callers. As a result, it is not uncommon for two completely unrelated parts of the kernel to subsequently operate on the same address range within a short time-frame. It also means that leaks of dynamically allocated ring 0 memory may disclose data processed by various components in the operating system. A more detailed analysis of the nature of data found in different memory regions can be found in Section 2.4 "Severity and impact on system security".

In the paragraphs below, we provide a brief overview of the allocators used in the Windows and Linux kernels and their most notable qualities.

Windows The core of the Windows kernel pool allocator is the `ExAllocatePoolWithTag` function [82], which may be used directly, or via several available wrappers: `ExAllocatePool`{∅, `Ex`, `WithQuotaTag`, `WithTagPriority`}. None of these routines reset the returned memory regions, neither by default, nor through any input flags. On the contrary, all of them have the following warning in their corresponding MSDN documentation entries:

> **Note** Memory that *function* allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

There are six primary pool types the callers can choose from: `NonPagedPool`, `NonPagedPoolNx`, `NonPagedPoolSession`, `NonPagedPoolSessionNx`, `PagedPool` and `PagedPoolSession`. Each of them has a designated region in the virtual address space, and so memory allocations may only be reused within the scope of their specific pool type. The reuse rate of memory chunks is very high, and zeroed out areas are generally only returned when no suitable entry is found in the lookaside lists, or the request is so large that fresh memory pages need to be mapped to facilitate it. In other words, there are currently hardly any factors preventing pool memory disclosure in Windows, and almost every such bug can be exploited to leak sensitive data from various parts of the kernel.

Linux The Linux kernel offers three main interfaces for dynamically allocating memory:

- `kmalloc` – a generic function used to allocate arbitrarily sized memory chunks (continuous in both virtual and physical address space), backed by the slab allocator.
- `kmem_cache_create` and `kmem_cache_alloc` – a specialized mechanism for allocating objects of a fixed size (e.g. structures), also backed by the slab allocator.
- `vmalloc` – a rarely used allocator returning regions that are not guaranteed to be continuous in physical memory.

By themselves, these functions provide no assurance that the allocated regions won't contain old, potentially sensitive data, making kernel heap memory disclosure feasible. However, there is a number of ways callers can request zeroed-out memory:

- The `kmalloc` function has a `kzalloc` counterpart, which makes sure that the returned memory is cleared.
- An optional `__GFP_ZERO` flag can be passed to `kmalloc`, `kmem_cache_alloc` and several other functions to achieve the same result.
- The `kmem_cache_create` routine accepts a pointer to an optional *constructor* function, called to pre-initialize each object before returning it to the requestor. The constructor may be implemented as a wrapper around `memset` to reset the given memory area.

We see the existence of these options as an advantage for kernel security, as they encourage developers to make conscious decisions, and enable them to express their intent of operating on a clean allocation at the time of requesting memory, instead of having to add extra `memset` calls in every such instance.

2.2.2 Fixed-sized arrays

A number of resources in operating systems can be accessed through their textual names. The variety of named resources is notably vast in Windows, including files and directories, registry keys and values, windows, fonts, atoms, and so forth. For some of them, the name length is limited by design, and is expressed by a constant such as `MAX_PATH` (260) or `LF_FACESIZE` (32). In such cases, kernel developers often simplify the code by declaring buffers of the maximum allowed size and copying them around as a whole (e.g. using the `sizeof` keyword) instead of operating only on the relevant part of the string. This is especially convenient if the strings are members of larger structures – such objects can be freely moved around in memory without worrying about the pointer management of any dependent allocations.

As can be expected, large buffers are rarely used to their full capacity, and the remaining storage space is often not reset. This may lead to particularly severe leaks of long, continuous areas of kernel memory. In the fictional example shown in Listing 5, a system call uses a `RtlGetSystemPath` function to load the system path into a local buffer, and if the call succeeds, all of the 260 bytes are passed to the caller, regardless of the actual length of the string.

```
NTSTATUS NTAPI NtGetSystemPath(PCHAR OutputPath) {
  CHAR SystemPath[MAX_PATH];
  NTSTATUS Status;

  Status = RtlGetSystemPath(SystemPath, sizeof(SystemPath));
  if (NT_SUCCESS(Status)) {
    RtlCopyMemory(OutputPath, SystemPath, sizeof(SystemPath));
  }

  return Status;
}
```

Listing 5: Memory disclosure via a partially initialized string buffer

The memory region copied back to user-mode in this example is illustrated in Figure 4.

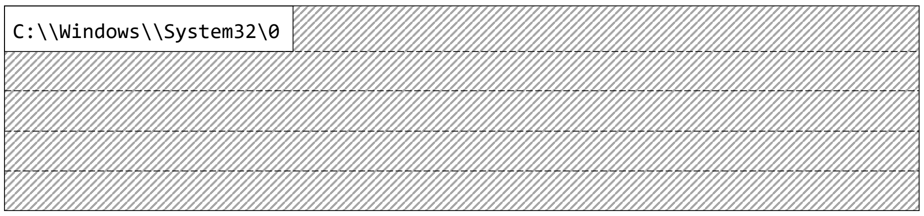

Figure 4: Memory layout of a partially initialized string buffer

A secure implementation should only return the requested path, with no extra padding bytes. Once again, this example demonstrates how the evaluation of the `sizeof` operator (used as a parameter to `RtlCopyMemory`) may be utterly inconsistent in relation to the actual amount of data the kernel is supposed to pass to userland.

2.2.3 Arbitrary syscall output buffer lengths

Most system calls accept pointers to user-mode output buffers together with the size of the buffers. In most cases, information about the size should only be used to determine if it is large enough to receive the syscall's output data, but should not otherwise influence how much memory is copied. However, we have observed cases where the kernel would try to fill every byte of the user's output buffer, disregarding the amount of actual data there is to copy. An example of such behavior is shown in Listing 6.

```
1   NTSTATUS NTAPI NtMagicValues(LPDWORD OutputPointer, DWORD OutputLength) {
2     if (OutputLength < 3 * sizeof(DWORD)) {
3       return STATUS_BUFFER_TOO_SMALL;
4     }
5
6     LPDWORD KernelBuffer = Allocate(OutputLength);
7
8     KernelBuffer[0] = 0xdeadbeef;
9     KernelBuffer[1] = 0xbadc0ffe;
10    KernelBuffer[2] = 0xcafed00d;
11
12    RtlCopyMemory(OutputPointer, KernelBuffer, OutputLength);
13    Free(KernelBuffer);
14
15    return STATUS_SUCCESS;
16  }
```

Listing 6: Memory disclosure via an arbitrary buffer output size

The purpose of the fictional system call is to provide the caller with three special 32-bit values, occupying a total of 12 bytes. While the buffer size sanity check in lines 2-4 is correct, the usage of the `OutputLength` argument should end at that. Knowing that the output buffer is large enough to store the result, the kernel could allocate a 12-byte buffer, fill it out accordingly and copy its contents to the user-mode location. Instead, the syscall requests an allocation with a user-controlled length and copies it back as a whole, even though there is only a fixed number of bytes that the caller needs. The remaining bytes are uninitialized and mistakenly disclosed to userland, as presented in Figure 5.

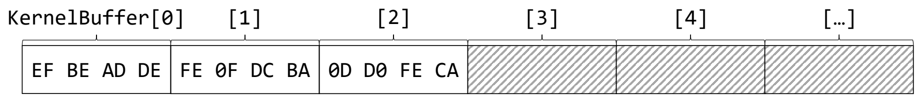

Figure 5: Memory layout of a memory allocation with a user-controlled size

The scheme discussed in this section is particularly characteristic to Windows. When implemented incorrectly, it may provide an attacker with an extremely useful memory disclosure primitive, for two reasons:

- An optimization frequently employed in Windows system calls is to use stack-based buffers for small user-defined buffer sizes, and pool allocations for larger ones. In combination with an infoleak bug, this could facilitate the disclosure of both kernel stack and pool memory through a single vulnerability.
- Being able to leak data from pool allocations of controlled lengths gives the attacker significantly more options as for the types of sensitive information they can acquire. As modern allocators tend to cache memory regions for

subsequent requests of the same length, it becomes possible to have the leaked allocations overlap with ones that had contained a specific kind of confidential data in the past.

As such, this is one of the most dangerous types of memory disclosure. It can be addressed by keeping track of the number of bytes written to the temporary kernel buffer, and passing only this amount of data back to the client.

2.3 Further contributing factors

In the previous section, we described some of the reasons why introducing information disclosure bugs to kernels is trivial, and how it can happen involuntarily. Here, we show the challenges behind identifying these bugs later in the development, which contributed to the lack of recognition of the problem for many years, with dozens of kernel infoleaks piling up in the Windows kernel.

Lack of visible consequences As a general rule, information disclosure flaws are more difficult to detect than memory corruption ones. The latter usually visibly manifest themselves in the form of crashes or other software misbehavior, especially when coupled with mechanisms such as AddressSanitizer, PageHeap or Special Pool. On the contrary, information disclosure doesn't cause any observable problems, and cannot be easily recognized by programmatic solutions. The confidentiality of any piece of information is highly subjective, and it takes a human to determine if the presence of some information in a specific security context is legitimate or not. On the other hand, with gigabytes of data being transferred from ring 0 to ring 3, from program memory to files on disk, from one machine to another through the network, and between a number of other security contexts, it is hardly possible to manually review all this traffic to find every instance of memory disclosure taking place. As a result, many bugs may actively leak data for years before they are noticed, if they are noticed at all.

The fact that infoleaks produce no tangible feedback to kernel developers also means that they cannot learn from their own mistakes, and may repeat the same insecure code patterns in a number of places if they are not aware of the bug class and are not actively trying to prevent it.

Leaks hidden behind system API Typical client applications implement their functionality using high-level system API, especially in Windows. The API is often responsible for converting input parameters into internal structures accepted by the system calls, and similarly converting the syscalls' output to a format understood by the caller. In the process, memory disclosed by the kernel may be discarded by the user-mode system library, and thus may never be passed back to the program. This further reduces the visibility of infoleaks, and decreases their chance of being noticed during regular software development. The situation is illustrated in Figure 6.

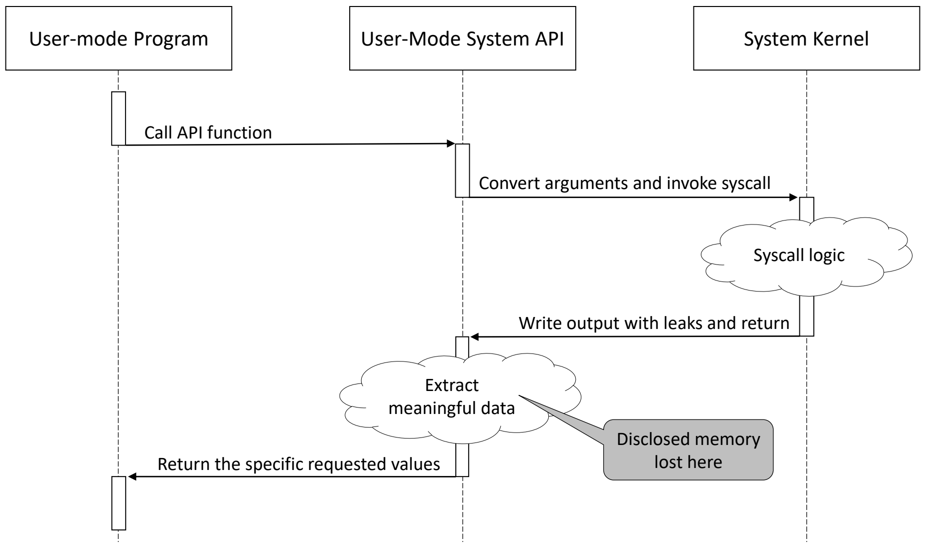
User-mode Program
User-Mode System API
System Kernel
Call API function
Convert arguments and invoke syscall
Syscall logic
Write output with leaks and return
Extract meaningful data
Disclosed memory lost here
Return the specific requested values

Figure 6: Leaked kernel memory discarded by user-mode API

2.4 Severity and impact on system security

Due to their very nature, kernel→user memory leaks are strictly local information disclosure bugs. There is no memory corruption or remote exploitation involved, and an attacker needs to be able to execute arbitrary code on the affected machine as a prerequisite. On the other hand, most disclosures are silent and don't leave any footprints in the system, so it is possible to trigger the bugs indefinitely, until the objective of the exploit is accomplished. The severity of each issue must be evaluated on a case-by-case basis, as it depends on the extent of the leak and the type of data that can be exposed.

In general, bugs of this class seem to be mostly useful as single links in longer local *elevation of privileges* exploit chains. The biggest secret-based mitigation currently in use is KASLR (*Kernel Address Space Layout Randomization*), and the locations of various objects in kernel address space are among the most frequently leaked types of data. A real-life example is a Windows kernel exploit found in the Hacking Team dump in July 2015 [40], which took advantage of a pool memory disclosure to derandomize the base address of the `win32k.sys` driver, subsequently used for the exploitation of another vulnerability. Coincidentally, the flaw was discovered around the same time by Matt Tait of Google Project Zero, and was later fixed in the MS15-080 bulletin as CVE-2015-2433 [75].

Stacks The distinction between stacks and heaps/pools is that stacks generally store information directly related to the control flow, such as addresses of kernel modules, dynamic allocations, stacks, and the secret values of stack cookies installed by exploit mitigations such as StackGuard on Linux [24] and `/GS` on Windows [83]. These are consistent pieces of information immediately useful for potential attackers who intend to combine them with memory corruption exploits. However, the variety of data they offer is limited[2], which leads us to believe that they don't present much value as standalone vulnerabilities.

Pools/Heaps Kernel pools and heaps contain addresses of executable images and dynamic allocations too, but they also include a range of data processed by various components in the system, such as disk drivers, file system drivers, network drivers, video drivers and so forth. This may allow attackers to effectively sniff the activity of privileged services and other users in the system, potentially revealing sensitive data which has value on its own, beyond only facilitating the exploitation of another bug. The problem of reliably leaking specific types of data from the kernel (e.g. file contents, network traffic or passwords) is still open and largely unexplored.

2.5 Prior research

Various degrees of work have been put to address memory disclosure in different operating systems. In the subsections below we outline the documented efforts undertaken for Windows and Linux in the areas of detection and mitigation.

2.5.1 Microsoft Windows

Detection There are very few publicly available sources discussing the problem in Windows before 2015. In July 2015, Matt Tait reported a disclosure of uninitialized pool memory through the `win32k!NtGdiGetTextMetrics` system call [75], which was also revealed a few weeks later to be known by the Hacking Team security firm [40]. To our best knowledge, this was the first piece of evidence that vulnerabilities of this type had been successfully used as part of 0-day local privilege escalation exploit chains for Windows.

Also in 2015, WanderingGlitch of HP's Zero Day Initiative was credited by Microsoft for the discovery of eight kernel memory disclosure vulnerabilities [78]. Some of them were later discussed by the researcher at the Ruxcon 2016 security conference, during a talk titled "Leaking Windows Kernel Pointers" [109].

Lastly, in June 2017 fanxiaocao and pjf of IceSword Lab (Qihoo 360) published a blog post titled "Automatically Discovering Windows Kernel Information Leak Vulnerabilities" [27], where they described the design of their tool, which they used to identify a total of 14 infoleaks in 2017 (8 of which collided with our findings). In many ways, the project was similar to Bochspwn Reloaded in how it poisoned kernel stack and pool allocations, and automatically detected leaks by examining all kernel$\rightarrow$userland memory writes. The key differences were that a virtual machine managed by VMware was used instead of a full

[2]As long as old heap pages are not reused for stacks without clearing, which was still the case in macOS in October 2017 [34], and in Linux until April 2018 [41].

CPU emulator, and memory taint tracking was not implemented. More information on this approach can be found in Section 5.5 "Taintless Bochspwn-style instrumentation". To our best knowledge, aside from Bochspwn Reloaded, this was the first attempt to automatically identify kernel memory disclosure in a closed-source operating system.

While not directly related to memory disclosure, it is also worth noting that in 2010/2011, several types of ring 0 addresses were found to be leaked through uninitialized return values of several `win32k` system calls [104, 51]. The problem was supposed to be mitigated in Windows 8 [42], but in 2015, Matt Tait spotted that the fix had been incomplete and discovered three further issues [74].

Mitigation One of the few generic mitigations we are aware of is that since June 2017, the Windows I/O manager resets the output memory regions for all buffered I/O operations handled by kernel drivers [39]. This change killed an entire class of infoleaks where the IOCTL handlers left uninitialized chunks of bytes in the buffer, or didn't initialize the buffer at all.

Another minor security improvement is the fact that in Visual Studio 15.5 and later, POD (*plain old data*) structures that are initialized at declaration using a "`= {0}`" directive are filled with zeros as a whole. This is different from the previous behavior, where the first member was set-by-value to 0 and the rest of the structure was reset starting from the second field, thus potentially leaking the contents of the padding bytes between the first and second member.

2.5.2 Linux

In contrast to Windows, the Linux community has been vocal about memory disclosure for many years, with the biggest spike of interest starting around 2010. Since then, a number of research projects have been devised, focusing on automatic detection of existing kernel infoleaks and on reducing or completely nullifying the impact of presumed, yet unknown issues. We believe that the gap in the current state of the art between Windows and Linux is primarily caused by the open-source nature of the latter, which enables easy experimentation with a variety of approaches – static, dynamic, and a combination of both.

Detection Throughout the last decade, there have been dozens of kernel infoleak fixes committed to the Linux kernel every year. According to Chen et al. [22], there were 28 disclosures of uninitialized kernel memory fixed in the period from January 2010 to March 2011. An updated study by Lu K. [44] shows that further 59 such vulnerabilities were reported between January 2013 and May 2016. A large portion of the findings can be attributed to a small group of researchers. For example, Rosenberg and Oberheide collectively discovered more than 25 memory disclosure vulnerabilities in Linux in 2009-2010 [25, 26, 35], mostly from the kernel stack. They subsequently demonstrated the usefulness of such disclosures in the exploitation of grsecurity/PaX-hardened Linux kernels in 2011 [37, 36]. Kulikov found over 25 infoleaks in 2010-2011 with manual analysis and Coccinelle [107]. Similarly, Krause identified and patched 21 kernel memory disclosure issues in 2013 [73], and more than 50 such bugs overall.

There are several tools readily available to detect infoleaks and other uses of uninitialized memory in Linux, mostly designed with kernel developers in mind. The most basic one is the `-Wuninitialized` compiler flag supported by both gcc and LLVM, capable of detecting simple instances of uninitialized reads within the scope of single functions. A more advanced option is the *kmemcheck* debugging feature [108], which can be considered a kernel-mode equivalent of Valgrind's memcheck. At the cost of a significant CPU and memory overhead, the dynamic checker detects all uses of uninitialized memory occurring in the code. The feature was recently removed from the mainline kernel [17, 76], as the project is now considered inferior to the new and more powerful *KernelAddressSanitizer* [12] and *KernelMemorySanitizer* checkers [19]. In the last few months, KMSAN coupled with the coverage-based *syzkaller* system call fuzzer [11] has identified 19 reads of uninitialized memory [7], including three leaks of kernel memory to userspace.

There have also been some notable efforts to use static analysis to detect Linux kernel infoleaks. In 2014-2016, Peiró et al. demonstrated successful use of model checking with the Coccinelle engine [3] to identify stack-based memory disclosure in Linux kernel v3.12 [97, 98]. The model checking was based on taint tracking objects in memory from stack allocation to `copy_to_user` calls, and yielded eight previously unknown vulnerabilities. In 2016, Lu et al. implemented a project called UniSan [45, 44] – advanced, byte-granular taint tracking performed at compile time to determine which stack and dynamic allocations could leak uninitialized memory to one of the external data sinks (user-mode memory, files and sockets). While the tool was primarily meant to mitigate infoleaks by clearing all potentially *unsafe* allocations, the authors randomly chose and analyzed a sample of about 20% of them (350 of about 1800), and reported 19 new vulnerabilities in the Linux and Android kernels as a result.

Finally, several authors have proposed a technique of multi-variant program execution to identify use of uninitialized memory. The basic premise of the approach is to concurrently run several replicas of the same software, capture their output and compare it. If all legitimate sources of entropy are virtualized to return stable data across all replicas, then any differences in the output may only be caused by leaking or using uninitialized memory. The non-determinism may originate from entropy introduced by ASLR, or from different marker bytes used to initialize new stack/heap allocations. The method was implemented for user-mode programs in the DieHard [21] and BUDDY [44] projects in 2006 and 2017, respectively. A similar approach was discussed by North in 2015 [96]. Lastly, authors of SafeInit [94] also stated that their tool was meant as a software hardening mechanism, but could be combined with a multi-variant execution system to achieve bug detection capability. The technique was extensively evaluated for client applications, but to our knowledge it hasn't been successfully implemented for the Linux kernel. In Section 5.4 "Differential syscall fuzzing", we present how a similar concept was shown to be effective in identifying infoleaks in a subset of Windows system calls.

Mitigation Generic mitigations against kernel memory disclosure generally revolve around zeroing old memory regions to prevent leftover data from being inherited by new, unrelated objects. The ultimate advantage of this method is that it addresses the problem on a fundamental level, completely eliminating the threat of uninitialized memory, and killing existing and future kernel infoleaks at once. Only a small fraction of memory allocated by the kernel is ever copied to user-mode; on the other hand, resetting all memory areas prior or after usage incurs a significant overhead. Finding the optimal balance between system performance and the degree of protection against memory disclosure is currently the main point of discussion.

In mainline kernel, the `CONFIG_PAGE_POISONING` and `CONFIG_DEBUG_SLAB` options can be set to enable the poisoning of all freed dynamic allocations with marker bytes. The result is that each piece of data is overwritten at the moment it is discarded by the caller, which prevents it from being leaked back to user-mode later on. Since all allocations are subject to poisoning, the options come with a considerable performance hit, and they don't protect stack allocations, which seem to constitute a majority of Linux kernel infoleaks.

The grsecurity/PaX project [4, 9] provides further hardening features. Setting the `PAX_MEMORY_SANITIZE` flag causes the kernel to erase memory pages and slab objects when they are freed, except for *low-risk* slabs whitelisted for performance reasons. Furthermore, the `PAX_MEMORY_STRUCTLEAK` option is designed to zero-initialize stack-based objects such as structures, if they are detected to be copied to userland. It may prevent leaks through uninitialized fields and padding bytes, but it is a relatively lightweight feature that may be subject to false negatives. A more exhaustive, but also more costly option is `PAX_MEMORY_STACKLEAK`, which erases the used portion of the kernel stack before returning from each system call. This eliminates any disclosure of stack memory shared between two subsequent syscalls, but doesn't affect leaks of data put on the stack by the vulnerable syscall itself. Currently, the *Kernel Self Protection Project* is making efforts to port the `STACKLEAK` feature to the mainline kernel [18, 38].

Other researchers have also proposed various variants of zeroing objects at allocation and deallocation times in the Linux kernel. *Secure deallocation* [23] (Chow et al., 2005) reduces the lifetime of data in memory by zeroing all regions at deallocation or within a short, predictable period of time. A prototype of the concept was implemented for Linux user-mode applications and the kernel page allocator. *Split Kernel* [43] (Kurmus and Zippel, 2014) protects the system from exploitation by untrusted processes by clearing entire kernel stack frames upon entering each hardened function. *SafeInit* [94] (Milburn et al., 2017) clears all stack and dynamic allocations before they are used in the code, to ultimately eliminate information leaks and use of uninitialized memory. UniSan [45] (Lu et al., 2016) reduces the overhead of SafeInit by performing advanced memory taint tracking at compile time, to conservatively determine which allocations are safe to be left without zero-initialization, while still clearing the remaining stack and heap-based objects.

As shown above, Linux has been subject to extensive experimentation in the area of data lifetime and kernel memory disclosure.

3 Bochspwn Reloaded – detection with software x86 emulation

Bochs [1] is an open-source IA-32 (x86) PC emulator written in C++. It includes emulation of the Intel x86 CPU, common I/O devices and a custom BIOS, making up a complete, functional *virtual machine* fully emulated in software. It can be compiled to run in a variety of configurations, and similarly it can correctly host most common operating systems, including Windows and GNU/Linux. In our research, we ran Bochs on Windows 10 64-bit as the host system.

Among many of the project's qualities, Bochs provides an extensive instrumentation API, which makes it a prime choice for performing DBI (*Dynamic Binary Instrumentation*) against OS kernels. At the time of this writing, there are 31 supported instrumentation callbacks invoked by the emulator on many occassions during the emulated computer's run time, such as:

- before starting the machine and after shutting it down,
- before and after the execution of each CPU instruction,
- on virtual and physical memory access,
- on exceptions and interrupts,
- on reading from and writing to I/O ports, and many more.

From a technical standpoint, the instrumentation is implemented in the form of C++ callback procedures built into the emulator executable. These callbacks may read and modify the CPU context and other parts of the execution environment (e.g. virtual memory), making it possible to inspect and alter the system execution flow in any desired way, as well as modify the processor's behavior and extend it with new functionality.

Basic documentation of the interface and the meaning of each particular callback can be found in the `instrument/instrumentation.txt` file in the Bochs source tree, and code examples are located in `instrument/example{0,1,2}`. The source code of the original Bochspwn tool from 2013 designed to identify double fetch issues in system kernels is also available for reference on GitHub [71].

In the following subsections, we describe how the core logic of memory disclosure detection and other ancillary features were implemented in the Bochs instrumentation.

3.1 Core logic – kernel memory taint tracking

The fundamental idea behind Bochspwn Reloaded is memory taint tracking performed over the entire kernel address space of the guest system. In this case, *taint* is associated with every kernel virtual address, and represents information about the initialization state of each byte residing in ring 0.

The high-level logic used to detect disclosure of uninitialized (tainted) kernel memory is as follows:

- Set taint on all new stack and heap/pool-based allocations.
- Remove taint from memory overwritten with constant values and registers.
- Remove taint from freed allocations (optionally).
- Propagate taint across memory during copy operations (`memcpy` calls).
- Identify infoleaks by verifying the taint of data copied from kernel-mode to user-mode addresses.

By implementing the above functionality, the instrumentation should be able to detect and signal most instances of uninitialized kernel memory being disclosed to userland. One of the obvious limitations of the approach is the fact that tainting is only applied to virtual memory and not CPU registers, which are effectively always considered untainted. This means that every time data in memory is copied indirectly through a register (e.g. with a sequence of `mov reg, [mem]; mov [mem], reg` instructions), the taint information is discarded in the process, potentially yielding undesired false-negatives.

The design decision to only taint memory and not the CPU context was primarily motivated by performance reasons. Instrumenting just the `memcpy` calls has low overhead; for example on x86 platforms, it only requires the special handling of the `rep movs{b,w,d,q}` instructions (as discussed in Section 3.1.5, "Taint propagation"). On the contrary, in order to propagate taint across registers, the instrumentation would have to cover all or most instructions operating on them, which would significantly slow down the already slow emulation of the guest system. Furthermore, Bochs doesn't provide any documented API to intercept register access, so more complex changes to the source code of the emulator would be necessary. Lastly, we believe that a majority of kernel infoleaks occur through large data structures and not primitive variable types, which suggests that the extent of false-negatives caused by the limited scope of the tainting should be negligible. Overall, our evaluation of the cost/benefit ratio of both methods seems to favor the more simplistic scheme.

Below, we discuss the implementation details of each part of the tainting mechanism used in Bochspwn Reloaded.

3.1.1 Shadow memory representation

The taint information of the guest kernel address space is stored in so-called *shadow memory*, a memory region in the Bochs process that maps each byte (or chunk of bytes) to the corresponding metadata. As the scope of the metadata and its low-level representation is different for 32-bit and 64-bit target systems, we address them separately in the following paragraphs.

Shadow memory in x86 In the default configuration of 32-bit versions of Windows, the kernel occupies addresses between `0x80000000` and `0xffffffff`, which makes a total of 2 GB. On Linux, in case of the 3G/1G memory split, the kernel is assigned 1 GB of the address space. Coupled with the fact that the Bochs emulator may be compiled as a 64-bit process, this means that as long as the amount of metadata is kept at a reasonable size, it is possible to statically allocate the shadow memory for the entire kernel address range.

In order to maximize the verbosity of Bochspwn Reloaded reports, we introduced a number of extra metadata information classes in addition to the taint bit, related to the memory allocation that each address belongs to:

- **size** – size of the memory allocation,
- **base address** – address of the start of the allocation,
- **tag/flags** – on Windows, the tag of the pool allocation; on Linux, the flags of the heap allocation,
- **origin** – address of the instruction that requested the allocation.

Each of the values is 32 bits wide. In total, the above fields together with the taint boolean consume 17 bytes. If a separate descriptor entry was allocated for each byte of the kernel address space, the overhead would be 34 GB for Windows guests and 17 GB for Linux. To optimize the memory usage, we increased the granularity of the above allocation-related metadata from 1 to 8 bytes, which reduced the effective overhead from a factor of 17 to 3. This was sufficient to run the instrumentation on a typical modern workstation equipped with a standard amount of RAM. A summary of the information classes is shown in Table 1.

Information class	Type	Granularity	Memory usage	
			Windows	Linux
Taint	`uint8`	1 byte	2 GB	1 GB
Allocation size	`uint32`	8 bytes	1 GB	512 MB
Allocation base	`uint32`	8 bytes	1 GB	512 MB
Allocation tag/flags	`uint32`	8 bytes	1 GB	512 MB
Allocation origin	`uint32`	8 bytes	1 GB	512 MB
Total			6 GB	3 GB

Table 1: Summary of metadata information classes stored for x86 guest systems

Shadow memory in x86-64 Representing taint information for 64-bit systems is more difficult, primarily because the address range subject to shadowing is as large as the user-mode address space of the Bochs emulator itself. Valid kernel memory addresses fall between `0xffff800000000000` and `0xffffffffffffffff`, which is a 128 terabyte area that cannot be mapped to a statically allocated region, both due to physical memory and virtual addressing limitations. As a result, this technical challenge called for a substantial rework of the metadata storage.

To start things off, we removed some of the less critical information related to memory allocations, specifically the *size*, *base address* and *tag/flags*. While certainly useful to have, we could still triage all potential reports without these information classes, and maintaining them for x86-64 targets would be a significant burden. However, we deemed the *allocation origin* important enough to stay, as it was often the only way to track the control flow back to the vulnerable code area, e.g. when universal interfaces (such as ALPC in Windows) were used to send data to user space. As allocating the data structure statically was no longer an option, we converted the `0x10000000`-item long array to a `std::unordered_map<uint64, uint64>` hash map container, semantically nearly identical to its 32-bit counterpart.

The last information class that needed to be considered was the taint itself. It was possible to also use a hash map in this case, but it was not optimal and would have significantly slowed the instrumentation down. Instead, we made use of the fact that the taint state for each kernel byte could be represented as a single bit. As a result, the taint information of the kernel address space was packed into bitmasks, thus mapping the `0x800000000000`-byte (128 TB) region into a `0x100000000000`-byte (16 TB) shadow memory. While an area of this size still cannot be allocated all at once, it can be reserved in the virtual address space. During the run time of the emulator, the specific pages accessed by the instrumentation are mapped on demand, resembling a mechanism known as memory overcommittment.

Memory overcommittment is not supported on Windows – our host system of choice – but it is possible to implement it on one's own using exception handling:

1. Reserve the overall shadow memory area with a `VirtualAlloc` API call and a `MEM_RESERVE` flag.

2. Set up an exception handler using the `AddVectoredExceptionHandler` function.

3. In the exception handler, check if the accessed address falls within the shadow memory, and exception code equals `EXCEPTION_ACCESS_VIOLATION`. If this is the case, commit the memory page in question and return with `EXCEPTION_CONTINUE_EXECUTION`.

With the modifications explained above, the shadow memory was successfully ported to work with 64-bit guest systems.

Double-tainting In addition to setting taint on allocations in the shadow memory, our instrumentation also fills the body of new memory regions with fixed marker bytes[3] – `0xaa` for heap/pools and `0xbb` for stack objects. While not essential for the correct functioning of the infoleak detection, it is a useful debugging feature.

First of all, the mechanism enables the instrumentation to verify the correctness of its own taint tracking by cross checking the information from two different sources. If a specific region is uninitialized according to the shadow memory, but in fact it no longer contains the marker bytes, this indicates that the memory was overwritten in a way our tool was not aware of (e.g. by a disk controller or another external device). In such cases, the instrumentation can correct the taint early on, instead of propagating the wrong information further in memory. Similarly, the behavior guarantees a nearly 100% true-positive ratio of the reported bugs, as they are verified both against the taint information and the guest virtual memory.

Furthermore, the markers are easily recognizable under kernel debuggers attached to the guest systems, which often aids in understanding the current system state and thus makes it easier to establish the root cause of the discovered bugs. Lastly, the mechanism may potentially expose other types of vulnerabilities in the process, such as use of uninitialized memory.

3.1.2 Tainting stack frames

Allocating memory from the stack is typically a platform-agnostic operation facilitated by the CPU architecture. On x86, it is achieved with a `sub esp` instruction, where the second operand is either an immediate value or another register. The equivalent `sub rsp` instruction is used on x86-64. This is the case in both Windows and Linux. In order to account for any unexpected code constructs that could also decrease `ESP` to allocate memory, we additionally instrumented the `add esp` and `and esp` instructions.

The overall logic of the stack tainting implemented in Bochspwn Reloaded is shown in pseudo-code in Listing 7. The *before execution* callback is used to detect instances of kernel instructions modifying the stack pointer. Later, after the instruction executes, a subsequent callback checks if `ESP` was decreased and if so, the entire memory region in the range of $\texttt{ESP}_{\texttt{new}} - \texttt{ESP}_{\texttt{old}}$ is tainted. The current instruction pointer is also saved, so that the memory can be traced back to the place where it was allocated in case it is detected to be disclosed to user-mode. This universal approach is effective for both Linux and a majority of the Windows kernel. The few remaining corner cases in Windows that require special handling are discussed in the next paragraph.

[3]Unless the caller explicitly requests a zeroed-out area, e.g. by passing the `__GFP_ZERO` flag to `kmalloc` in Linux.

```
void bx_instr_before_execution(CPU cpu, instruction i) {
  if (!cpu.protected_mode ||
      !os::is_kernel_address(cpu.eip) ||
      !os::is_kernel_address(cpu.esp)) {
    return;
  }

  if (i.opcode == SUB || i.opcode == ADD || i.opcode == AND) {
    if (i.op[0] == ESP) {
      globals::esp_changed = true;
      globals::esp_value = cpu.esp;
    }
  }
}

void bx_instr_after_execution(CPU cpu, instruction i) {
  if (globals::esp_changed && cpu.esp < globals::esp_value) {
    set_taint(/*from=  */cpu.esp,
              /*to=    */globals::esp_value - 1,
              /*origin=*/cpu.eip);
  }

  globals::esp_changed = false;
}
```

Listing 7: Pseudo-code of the stack tainting logic

Microsoft Windows One system-specific assembly code construct not covered by the previously described logic is the implementation of the `__chkstk` built-in function in 32-bit builds of Windows. The routine is equivalent to `alloca` in that it is used for requesting dynamic allocations from the stack. The relevant part of the code is shown in Listing 8; the highlighted `xchg eax, esp` instruction is used to save the updated stack pointer in `ESP`. In order to account for this non-standard way of modifying `ESP`, we included `BX_IA_XCHG_ERXEAX` in the list of opcodes possibly manifesting a stack allocation.

As `__chkstk` is a generic function, saving its address doesn't reveal the actual requestor of the memory region. In order to work around this, our instrumentation reads the return address from `[EAX]` after the `xchg` instruction executes, and uses it to identify the creator of the allocation.

There is also another circumstance when the instruction directly modifying `ESP` cannot be used as the allocation's origin. The `__SEH_prolog4` and `__SEH_prolog4_GS` procedures are used in 32-bit Windows to create stack frames for functions with exception handlers, including most top-level syscall entry points. A shortened version of the `__SEH_prolog4` function assembly is shown in Listing 9. To handle this case, Bochspwn Reloaded obtains the address of the procedure caller from `[EBP-8]`, and uses it as the origin of the stack frame.

```
.text:00548D94                 public __chkstk
.text:00548D94 __chkstk        proc near
.text:00548D94
[...]
.text:00548DA8
.text:00548DA8 cs10:
.text:00548DA8                 cmp     ecx, eax
.text:00548DAA                 jb      short cs20
.text:00548DAC                 mov     eax, ecx
.text:00548DAE                 pop     ecx
.text:00548DAF                 xchg    eax, esp
.text:00548DB0                 mov     eax, [eax]
.text:00548DB2                 mov     [esp+0], eax
.text:00548DB5                 retn
.text:00548DB6
[...]
.text:00548DBD __chkstk        endp
```

Listing 8: Built-in __chkstk function on 32-bit versions of Windows

```
.text:00557798 __SEH_prolog4   proc near
.text:00557798
.text:00557798 arg_4           = dword ptr  8
.text:00557798
.text:00557798                 push    offset __except_handler4
[...]
.text:005577AC                 lea     ebp, [esp+8+arg_4]
.text:005577B0                 sub     esp, eax
.text:005577B2                 push    ebx
[...]
.text:005577D3                 lea     eax, [ebp-10h]
.text:005577D6                 mov     large fs:0, eax
.text:005577DC                 retn
.text:005577DC __SEH_prolog4   endp
```

Listing 9: Built-in __SEH_prolog4 function on 32-bit versions of Windows

3.1.3 Tainting heap/pool allocations

Contrary to stack frames and automatic objects, detecting and tainting dynamic allocations is a highly system-specific task, which must be implemented for each tested OS dedicatedly. As a general rule, the instrumentation should intercept the addresses of all newly allocated regions and their lengths, optionally together with further information such as the origin, tag/flags etc. The specifics of the Windows and Linux kernel allocators are outlined in the paragraphs below.

Microsoft Windows As noted in Section 2.2.1 "Memory reuse in dynamic allocators", the core of the Windows kernel pool allocator is the `ExAllocatePoolWithTag` function [82]. While there are many available wrappers which offer extended functionality, all of them eventually call into `ExAllocatePoolWithTag` to request the dynamic memory. Thanks to this design, it is sufficient to hook into this single function to obtain information about nearly all dynamic allocations taking place in the kernel.

In 32-bit versions of Windows, the function follows the *stdcall* calling convention, which means that the arguments are stored on the stack, and the return value is passed back via the `EAX` register. This control flow is convenient for our instrumentation, as it allows us to set a single "breakpoint" on the `RET` instruction(s) at the end of the allocator, and every time it is hit, we can read both the details of the request (allocation origin, size and tag) and the address of the allocated memory. At that point, the instrumentation marks the entire region as tainted in the shadow memory.

The corresponding implementation for 64-bit builds of Windows is more complex, due to the fact that parameters to `ExAllocatePoolWithTag` are passed through the `RCX`, `RDX` and `R8` registers. These registers are considered volatile, which means that by the time the allocator returns, they may be modified and no longer contain the input values. In other words, there isn't a single point in the control flow where both the request information and the allocation address are guaranteed to be known at the same time. To address the problem, we hook into the allocator twice – in the prologue and epilogue of the function. In the first hook, we read the allocation origin and length, and save them in a hash map keyed by `RSP`. In the second one, we read the allocation address from `RAX`, load the request information from the hash map (the stack pointer is the same when entering and leaving the function), and taint the memory accordingly.

One corner case that requires special attention are alternate kernel allocators used by specific subsystems in the kernel. Even though they are typically based on `ExAllocatePoolWithTag`, they may implement additional optimizations such as caching, which bypass our taint tracking algorithm. One example of such an allocator is the `win32k!AllocFreeTmpBuffer` function, whose pseudo-code is illustrated in Listing 10. The function maintains a global pointer to a long-lasting `0x1000`-byte allocation, which is used to satisfy requests whenever it is not currently in use. As the memory region is allocated only once and never freed, but it is reused by multiple parts of the graphic subsystem, it is not subject to proper memory disclosure detection. The solution to this problem is to modify

```
PVOID AllocFreeTmpBuffer(SIZE_T Size) {
  PVOID Result;

  if (Size > 0x1000 ||
      (Result = InterlockedExchange(gpTmpGlobalFree, NULL)) == NULL) {
    Result = AllocThreadBufferWithTag(Size, 'pmTG');
  }

  return Result;
}
```

Listing 10: Pseudo-code of the `win32k!AllocFreeTmpBuffer` allocator

`AllocFreeTmpBuffer` to always call into `AllocThreadBufferWithTag`, or to change the value of the pointer under `gpTmpGlobalFree` to `NULL`. Both actions have the same outcome and are easily achieved with the WinDbg debugger attached to the guest system; the latter has the advantage that it works with 64-bit builds of Windows with PatchGuard enabled.

Linux Compared to Windows, intercepting all kernel dynamic allocations in Linux from the level of DBI is relatively complex for several reasons. First of all, the x86 kernel is compiled with the `regparm=3` option, which causes the first three arguments of each function to be passed through volatile registers instead of the stack. As a result, the values of these parameters are no longer available when the function returns – if our instrumentation requires access to both the arguments and return value, it needs to hook into the function in question twice, in the prologue and epilogue(s). Furthermore, there are three distinct allocators (`kmalloc`, `vmalloc`, `kmem_cache_alloc`), and some of them have many entry-points, e.g. `__kmalloc`, `kmalloc_order` and `__kmalloc_track_caller`. Lastly, in case of `kmem_cache`, the allocation size must be saved during the creation of the cache, as it is not explicitly provided later while requesting memory. If the cache has a constructor specified, then the taint must be applied when the constructor is called, and not when the memory is returned to the caller.

In response to the issues raised above, the Linux-specific part of our instrumentation installed a number of hooks in the emulated kernel:

- `kmalloc` and `vmalloc`
  - Prologue: save the `size` and `flags` arguments.
  - Epilogue: set taint on the allocated memory.
- `kfree`, `vfree` and `kmem_cache_free` – untaint freed allocations.

- `kmem_cache_create`
  - Prologue: save the cache size and the constructor function pointer.
  - Epilogue: if the function succeeded, save the address of the newly created cache and set a breakpoint on the constructor function, if present.
- `kmem_cache_destroy` – remove the cache from the internal structures and clear the breakpoint on the cache's constructor, if present.
- `kmem_cache_alloc`
  - Prologue: save the `cache` and `flags` arguments.
  - Epilogue: set taint on the allocated memory.
- Dynamically detected cache constructors – set taint on the memory region received in the argument.

By intercepting the kernel execution in those locations, it was possible to apply taint tracking to dynamic allocations being requested during the run time of the Linux kernel.

3.1.4 Clearing taint

By principle, Bochspwn Reloaded untaints memory every time it is overwritten by any instruction which is not part of a `memcpy` operation. This includes writes with constant values and registers, which by themselves are not subject to taint tracking. Therefore, virtually any write to memory marks it as initialized as long as it is not receiving existing data from another source.

Optionally, memory can also be untainted upon being freed, for example on a `add esp` instruction increasing the address of the stack pointer, or when a `ExFreePoolWithTag` / `kfree` function is called to destroy a dynamic object. However, doing so is not required, and Solar Designer suggested [102] that it could be beneficial to re-taint freed allocations in order to potentially detect use-after-free and similar vulnerabilities. In our testing, we untainted freed allocations in 32-bit Windows and Linux guests, but didn't handle any `free`-like functions in 64-bit Windows.

3.1.5 Taint propagation

In Bochspwn Reloaded, taint is propagated when the instrumentation recognizes that data is copied between two kernel-mode locations in the guest memory, i.e. a `memcpy` operation is in progress. While primitive variables are out of scope due to the fact that CPU registers are not subject to taint tracking, it is paramount for the tool to effectively detect the copying of larger memory blobs.

On the x86 and x86-64 platforms, the instruction dedicated to copying continuous memory blocks is `rep movsb`[4], which moves `ECX` bytes at address `DS:ESI` to address `ES:EDI` (equivalent 64-bit registers are used on x86-64). It is very frequently used in kernels, both as part of the standard library `memcpy` implementation and as an inlined form of the function. From the perspective of instrumentation, the instruction is very convenient, as it specifies the *source*, *desination* and *length* of the copy all at the same time. This makes it trivial to propagate taint in the shadow memory, and is the main reason why taint propagation in our tool was built around the special handling of `rep movs`.

In the paragraphs below, we explain how practical the core idea was with regards to the actual binary code of each of the tested systems, and what further steps we took to maximize the scope of the detected memory copying activity.

Microsoft Windows In Windows, there are generally four functions which can be used to copy data in memory: `memcpy`, `memmove`, `RtlCopyMemory` and `RtlMoveMemory`.

On 32-bit builds of the system, `RtlCopyMemory` is not an actual function, but merely a macro around `memcpy`. The `RtlMoveMemory` routine is based exclusively on `rep movsb` and `rep movsd`, which makes it fully compatible with our taint tracking approach. Finally, `memcpy` and `memmove` are separate functions but have exactly the same code, so they can be considered as one. They are also *mostly* based on the desired `rep movs` instruction, with the exception of several conditions:

1. the source and destination buffers overlap,
2. the destination pointer is not 4-byte aligned,
3. the copy length is below 32 bytes,
4. the copy length is not 4-byte aligned.

Practically speaking, we believe that conditions (1) and (2) can be safely disregarded, as they occur very rarely during normal system execution and are unlikely to affect our infoleak detection performance. Condition (4) is also negligible, as it is similarly rare and only affects the copying of the trailing 1-3 bytes. Condition (3) is the most undesired one, as it may lead to the loss of taint corresponding to numerous small objects in the kernel address space.

In order to address the problem, it is possible to patch the `memcpy` and `memmove` functions in the WinDbg debugger attached to the guest system, such that they use `rep movs` regardless of the length of the operation. On binary level, it is a matter of modifying a single byte in the functions' body, changing the operand of a `cmp` instruction from `0x08` to `0x00`, thus nullifying the condition check. The relevant assembly code from the `nt!memcpy` function found in Windows 7 is shown in Listing 11. Thanks to the patch, the `rep movsd` instruction at address `0x439293` is always reached.

[4]The corresponding `rep movsw`, `rep movsd` and `rep movsq` variants operate on units of 2, 4 and 8 bytes respectively.

```
.text:00439260          _memcpy         proc near
.text:00439260
.text:00439260          arg_0           = dword ptr  8
.text:00439260          arg_4           = dword ptr  0Ch
.text:00439260          arg_8           = dword ptr  10h
.text:00439260
.text:00439260 55                       push    ebp
.text:00439261 8B EC                    mov     ebp, esp
[...]
.text:00439278 3B F8                    cmp     edi, eax
.text:0043927A 0F 82 7C+                jb      CopyDown
.text:00439280
.text:00439280          CopyUp:
.text:00439280 F7 C7 03+                test    edi, 3
.text:00439286 75 14                    jnz     short CopyLeadUp
.text:00439288 C1 E9 02                 shr     ecx, 2
.text:0043928B 83 E2 03                 and     edx, 3
.text:0043928E 83 F9 08                 cmp     ecx, 8
.text:00439291 72 29                    jb      short CopyUnwindUp
.text:00439293 F3 A5                    rep movsd
.text:00439295 FF 24 95+                jmp     ds:off_4393AC[edx*4]
```

Listing 11: Patched `memcpy` function prologue

It is important to note that both `memcpy` and `memmove` functions should be modified to achieve the best result. Moreover, depending on the specific Windows version, each driver may include its own copies of these functions. In our testing, we patched the code in the two most important modules – `ntoskrnl.exe` and `win32k.sys` – where applicable. Alternatively, one could also entirely overwrite the memory copy routines to exclusively use the `rep movsb` instruction, or implement the patching logic in the Bochs instrumentation instead of performing it manually in a kernel debugger.

On 64-bit Windows builds, it is more difficult to apply the previously discussed logic. Contrary to their 32-bit counterparts, the memory copy functions no longer use `rep movs` in any form, but instead they are optimized with the usage of `mov` and equivalent SSE instructions (`movdqu`, `movntdq` etc.). These instructions move memory indirectly through registers, and are therefore not compliant with our taint tracking logic. Example excerpts from the `nt!memcpy` 64-bit assembly code on Windows 7 and 10 are shown in Listings 12 and 13.

As we had already decided against taint tracking of CPU registers, which was the only generic solution to the problem, we adopted a more implementation-specific concept to dynamically detect all instances of the `memcpy` functions at run time, and propagate taint based on their arguments. In the context of this idea, it is convenient that in Windows x64, the four aforementioned memory copy functions are all handled by a single implementation, as illustrated in Listing 14. Furthermore, the binary representation of the functions' code in all

```
.text:0000000140095720 mcpy90:
.text:0000000140095720      mov     r9, [rdx+rcx]
.text:0000000140095724      mov     r10, [rdx+rcx+8]
.text:0000000140095729      movnti  qword ptr [rcx], r9
.text:000000014009572D      movnti  qword ptr [rcx+8], r10
.text:0000000140095732      mov     r9, [rdx+rcx+10h]
.text:0000000140095737      mov     r10, [rdx+rcx+18h]
.text:000000014009573C      movnti  qword ptr [rcx+10h], r9
.text:0000000140095741      movnti  qword ptr [rcx+18h], r10
.text:0000000140095746      mov     r9, [rdx+rcx+20h]
.text:000000014009574B      mov     r10, [rdx+rcx+28h]
.text:0000000140095750      add     rcx, 40h
.text:0000000140095754      movnti  qword ptr [rcx-20h], r9
.text:0000000140095759      movnti  qword ptr [rcx-18h], r10
.text:000000014009575E      mov     r9, [rdx+rcx-10h]
.text:0000000140095763      mov     r10, [rdx+rcx-8]
.text:0000000140095768      dec     eax
.text:000000014009576A      movnti  qword ptr [rcx-10h], r9
.text:000000014009576F      movnti  qword ptr [rcx-8], r10
.text:0000000140095774      jnz     short mcpy90
```

Listing 12: Example block of the `nt!memcpy` function on Windows 7 64-bit

```
.text:0000000140189700 lcpy40:
.text:0000000140189700      movdqu  xmm0, xmmword ptr [rdx+rcx]
.text:0000000140189705      movdqu  xmm1, xmmword ptr [rdx+rcx+10h]
.text:000000014018970B      movntdq xmmword ptr [rcx], xmm0
.text:000000014018970F      movntdq xmmword ptr [rcx+10h], xmm1
.text:0000000140189714      add     rcx, 40h
.text:0000000140189718      movdqu  xmm0, xmmword ptr [rdx+rcx-20h]
.text:000000014018971E      movdqu  xmm1, xmmword ptr [rdx+rcx-10h]
.text:0000000140189724      movntdq xmmword ptr [rcx-20h], xmm0
.text:0000000140189729      movntdq xmmword ptr [rcx-10h], xmm1
.text:000000014018972E      dec     eax
.text:0000000140189730      jnz     short lcpy40
```

Listing 13: Example block of the `nt!memcpy` function on Windows 10 64-bit

```
.text:0000000140095600 ; Exported entry 1365. RtlCopyMemory
.text:0000000140095600 ; Exported entry 1541. RtlMoveMemory
.text:0000000140095600 ; Exported entry 2063. memcpy
.text:0000000140095600 ; Exported entry 2065. memmove
.text:0000000140095600
.text:0000000140095600 ; Attributes: library function
.text:0000000140095600
.text:0000000140095600                 public memmove
.text:0000000140095600 memmove         proc near
.text:0000000140095600
.text:0000000140095600                 mov     r11, rcx
.text:0000000140095603                 sub     rdx, rcx
.text:0000000140095606                 jb      mmov10
.text:000000014009560C                 cmp     r8, 8
```

Listing 14: Shared implementation of memory copy functions in 64-bit Windows

kernel images across the system is the same, meaning that we could successfully identify all copies of `memcpy` in memory by recognizing a single unique signature of the routine's prologue. In our testing, we used the first 16 bytes of the function code, which corresponded to the four initial assembly instructions. Such a signature proved to uniquely identify the procedure in question, enabling our tool to track the memory taint on 64-bit Windows platforms.

One currently unresolved problem is the fact that with each newer version of Windows, an increasing number of `memcpy` instances with constant length are compiled as inlined sequences of `mov` instructions, instead of `rep movs` or direct calls into the library function. This is clearly visible in the numbers of references to the function – on Windows 7 64-bit (January 2018 patch), the `win32k.sys` module calls into `memcpy` at 1133 unique locations in the code. However, the combined drivers on Windows 10 Fall Creators Update 64-bit (`win32k.sys`, `win32kbase.sys` and `win32kfull.sys`) only invoke the function 696 times. The remaining instances were replaced with `mov` instructions located directly in the client functions, causing our tool to lose track of a large part of the kernel memory taint. We hope that the issue was partially mitigated by the fact that we performed the testing against Windows 7 and 10, so bugs dating back to Windows 7 should have been successfully detected and fixed in both versions, even if the reduced effectiveness of the taint tracking would prevent their discovery on Windows 10. Nonetheless, this circumstance consistutes a significant problem in the current scheme of the Bochspwn Reloaded project.

```
175  /*
176   *  No 3D Now!
177   */
178
179  #ifndef CONFIG_KMEMCHECK 0
180
181  #if (__GNUC__ >= 4)
182  #define memcpy(t, f, n) __builtin_memcpy(t, f, n)
183  #else
184  #define memcpy(t, f, n)       \
185    (__builtin_constant_p((n))    \
186     ? __constant_memcpy((t), (f), (n)) \
187     : __memcpy((t), (f), (n)))
188  #endif
189  #else
190  /*
191   * kmemcheck becomes very happy if we use the REP instructions
        unconditionally,
192   * because it means that we know both memory operands in advance.
193   */
194  #define memcpy(t, f, n) __memcpy((t), (f), (n))
195  #endif
```

Listing 15: Patch applied to `arch/x86/include/asm/string_32.h` to redirect `memcpy` to the Bochspwn-compliant `__memcpy` in line 194

Linux Linux being an open-source kernel, it gives us full control over how `memcpy` is compiled. Following brief experimentation, we determined that only minor code modifications were necessary to make it compliant with our tool, as the function's assembly code was largely influenced by the kernel configuration flags. More specifically, we set the `CONFIG_X86_GENERIC` option to `y` and `CONFIG_X86_USE_3DNOW` to `n`, and applied the patch shown in Listing 15 to unconditionally redirect `memcpy` invocations to the `__memcpy` function comprising of the `rep movsd` and `rep movsb` instructions. These actions were sufficient to ensure that the resulting memory-copying assembly worked with the taint propagation mechanism used in our instrumentation.

3.1.6 Bug detection

Bug detection in Bochspwn Reloaded may be considered a part of the taint propagation functionality. Whenever the instrumentation detects a kernel→kernel data copy, it propagates the corresponding taint in the shadow memory. When a kernel→userland copy takes place, the tool checks the source memory region for uninitialized bytes and if any are found, a vulnerability is reported. As a result, detection of memory disclosure is an extension of taint propagation, and works correctly as long as `memcpy`-like operations are properly instrumented.

There are no system-specific considerations related to bug detection in Windows. In Linux, we additionally implemented support for identifying information leaks through simple variable types, and more general detection of use of uninitialized memory. Both efforts are documented in the paragraphs below.

Leaks through primitive types in Linux In Linux, there are two interfaces facilitating the writing of data from the kernel into user space – `copy_to_user` and `put_user`. The `copy_to_user` function is an equivalent of `memcpy`, and is subject to the regular bug detection logic if the kernel is compiled with the `CONFIG_X86_INTEL_USERCOPY` option set to `n`. On the other hand, `put_user` is designed to allow the kernel to write values of simple types into ring 3, such as characters, integers or pointers. It uses direct pointer manipulation, and at the binary level, the data in question is copied through registers, so it is not subject to kernel infoleak detection based on the `rep movs` instruction. Considering that `put_user` is used extensively in the Linux kernel, and that we had the power to recompile the software to adjust it to our needs, we implemented an additional bug detection mechanism dedicated to `put_user`, which required changes in the source code of both the Linux kernel and the Bochs instrumentation.

The main problem related to the sanitization of data written with `put_user` is the fact that it is passed through value and not reference (pointer). Consequently, the first argument of the macro may be a constant, variable, structure/union field, array item, function return value, or an expression involving components of any of the above types. Therefore, it is not clear which particular memory region should be sanitized in each specific case, and it is difficult to isolate only the *simple* cases on the level of either kernel code or instrumentation.

While most CPU architectures supported by Linux have their own implementation of the `put_user` macro, there is also a generic version declared in `include/asm-generic/uaccess.h`. Listing 16 shows the body of `__put_user`, an internal macro which sits at the core of `put_user`. As we can see in line 147, the expression passed by the caller to be written to user space is evaluated and stored in a local, helper variable called `__x`. While analyzing the code, we decided to sanitize all memory reads executed as part of the expression evaluation, as long as they occured in the context of the current function (i.e. not in nested function calls). To that end, we modified the macro to wrap the initialization of the `__x` variable with two assembly instructions – `prefetcht1` and `prefetcht2`. The diff of the change is presented in Listing 17. As a result, all accesses to memory that was passed to user-mode through `put_user` were placed between the two artificially inserted instructions. Examples of disassembled code snippets from a Linux kernel compiled in this manner are shown in Listing 18.

In Bochs, the `prefetcht1` and `prefetcht2` instructions aren't emulated with any dedicated logic, but are instead handled by the `BX_CPU_C::NOP` method. This makes them prime candidates to be used as *hypercalls* – special opcodes that don't have any effect on the execution of the guest system, but are used to communicate with the emulator. In this case, our instrumentation detects the execution of `prefetcht1` and treats it as a signal to start sanitizing all

```
#define __put_user(x, ptr) \
({                                                              \
        __typeof__(*(ptr)) __x = (x);                           \
        int __pu_err = -EFAULT;                                 \
        __chk_user_ptr(ptr);                                    \
        switch (sizeof (*(ptr))) {                              \
        case 1:                                                 \
        case 2:                                                 \
        case 4:                                                 \
        case 8:                                                 \
                __pu_err = __put_user_fn(sizeof (*(ptr)),       \
                                         ptr, &__x);            \
                break;                                          \
        default:                                                \
                __put_user_bad();                               \
                break;                                          \
         }                                                      \
        __pu_err;                                               \
})
```

Listing 16: Declaration of the generic `put_user` macro (`include/asm-generic/uaccess.h`)

```
#define __put_user(x, ptr) \
({                                                              \
        __typeof__(*(ptr)) __x = (x);                           \
        int __pu_err = -EFAULT;                                 \
        __chk_user_ptr(ptr);                                    \
        __asm("prefetcht1 (%eax)");                             \
        __x = (x);                                              \
        __asm("prefetcht2 (%eax)");                             \
        switch (sizeof (*(ptr))) {                              \
```

Listing 17: Changes applied to the `put_user` macro

```
.text:C1027F72                prefetcht1 byte ptr [eax]
.text:C1027F75                mov     eax, [ebp+var_B4]
.text:C1027F7B                mov     [ebp+var_AC], eax
.text:C1027F81                prefetcht2 byte ptr [eax]
[...]
.text:C1035910                prefetcht1 byte ptr [eax]
.text:C1035913                mov     eax, [ebp+var_14]
.text:C1035916                mov     edx, edi
.text:C1035918                call    getreg
.text:C103591D                mov     [ebp+var_10], eax
.text:C1035920                prefetcht2 byte ptr [eax]
[...]
.text:C1071AD7                prefetcht1 byte ptr [eax]
.text:C1071ADA                mov     edx, [ebp+var_1C]
.text:C1071ADD                shl     edx, 8
.text:C1071AE0                or      edx, 7Fh
.text:C1071AE3                mov     [ebp+var_10], edx
.text:C1071AE6                prefetcht2 byte ptr [eax]
```

Listing 18: Instances of the compiled `put_user` macro with added marker instructions

kernel-mode memory references at the current value of `ESP`. If any uninitialized memory is read while operating in this mode, the potential bug is reported in the same way as any typical kernel infoleak. Accordingly, the `prefetcht2` instruction disables the *strict sanitization* mechanism.

Thanks to this design, Bochspwn Reloaded was capable of detecting references to uninitialized variables passed – directly or indirectly, as part of larger expressions – to the `put_user` macro. The approach is not free from occasional false positives, as the uninitialized memory may not always be propagated to user space (e.g. if it is an unused function argument). However, bug candidates flagged by the tool can be manually investigated to determine the root cause and severity of the findings, which is facilitated by the open-source nature of the Linux kernel. Overall, we consider it an effective way of instrumenting `put_user` calls, which seems an otherwise difficult task for dynamic binary instrumentation.

Use of uninitialized memory in Linux As discussed in Section 3.5.2, we initially didn't have much success with the kernel infoleak detection used against Linux. In order to test the correctness of the taint tracking and to generate more output to analyze, we extended the scope of detection to all reads of uninitialized memory. On a technical level, this was achieved by enabling the strict sanitization described in the previous section for the entirety of the kernel execution, and not just for blocks of code between the `prefetcht` instructions. In this mode, every unique read of uninitialized memory was flagged as a potential bug.

Considering that our instrumentation didn't recognize if the uninitialized data had any influence on the kernel control flow, the output logs included a number of false-positives where leftover memory was read (e.g. while being copied), but never actually used in a meaningful way. On the upside, the volume of the reports turned out to be manageable, and the false-positives were relatively easy to filter out upon brief analysis of the kernel code.

It is also important to note that while uses of uninitialized memory are typically real bugs, they are often functional and not security problems. For example, a majority of the issues identified by Bochspwn Reloaded in Linux had very little to no security impact. Nonetheless, proposed patches for all discovered bugs were submitted and subsequently accepted by the kernel developers.

3.2 Ancillary functionality

The core capabilities of the project discussed above are designed to enable effective detection of kernel infoleaks, but provide little extra information about the context of the bugs or the overall system state. In order to produce verbose reports that could be used to deduplicate vulnerabilities and quickly understand their root cause and impact, we implemented a series of additional features in the instrumentation. While they are not critical to the correct functioning of the tool, they have proved highly useful while analyzing and reproducing the identified flaws. The technical details behind these ancillary mechanisms are explained in the sections below.

3.2.1 Keeping track of kernel modules

Keeping track of kernel modules is the first step to symbolizing raw addresses from the guest system. The most important modules to track are `ntoskrnl.exe` in Windows and `vmlinux` in Linux. Based on their location in memory, the instrumentation can obtain the addresses of all other drivers loaded in the system.

One reliable way to acquire the base addresses of these core images is to intercept all writes to Model-Specific Registers (MSR) in search of addresses residing at known offsets inside of the desired kernel executables. An example of such MSRs are registers storing the addresses of system call entrypoints, used to transfer control flow upon the execution of `SYSENTER` and `SYSCALL` instructions. In protected mode, the register in question is `0x176` (`SYSENTER_EIP_MSR`), while in long mode, it is `0xC0000082` (`IA32_LSTAR`).

There are several advantages of using MSRs to determine the kernel base:

- The system call interface is initialized early during system boot.
- The syscall entrypoints are located at fixed offsets in the kernel images.
- Bochs provides a `BX_INSTR_WRMSR` instrumentation callback which receives notifications about all MSR writes taking place in the emulated system.

Once the base address of the primary image is established, listing other loaded modules is a matter of traversing through simple system-specific linked lists of driver descriptors in the guest memory. In Bochspwn Reloaded, the traversing is performed every time the instrumentation encounters an address that cannot be associated with any of the currently known modules.

In Windows, a static `nt!PsLoadedModuleList` variable points to the head of a doubly-linked list consisting of `LDR_MODULE` structures. In Linux, the corresponding head pointer is named `modules`, and the list is made of `module` structures. In both cases, each such structure describes a single kernel driver, including its name, base address and size in memory. The layouts of these lists are illustrated in Figures 7 and 8.

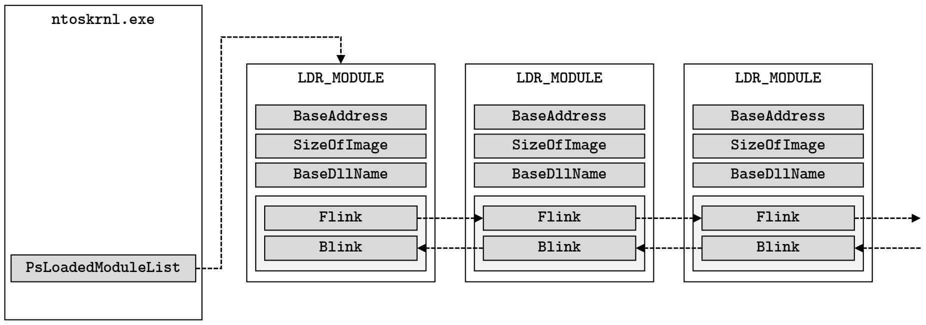

Figure 7: Windows kernel module linked list layout

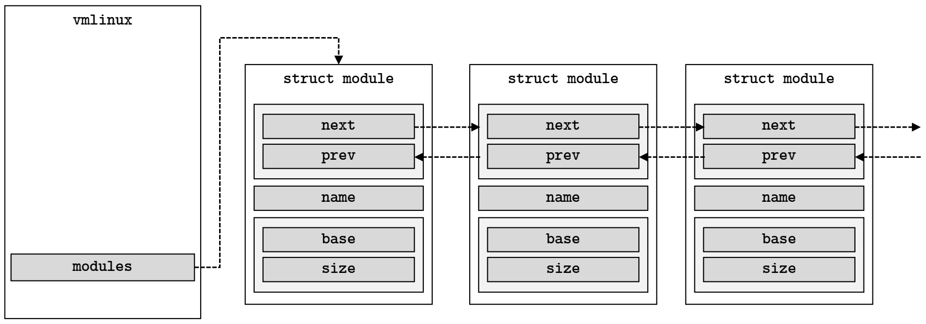

Figure 8: Linux kernel module linked list layout

3.2.2 Unwinding stack traces

Collecting full stack traces upon detecting kernel infoleaks helps both deduplicate the bugs (to avoid flooding logs with multiple instances of the same issue), and understand how the execution flow reached the affected code. Depending on the bitness of the guest system, the goal can be achieved in different ways.

On 32-bit builds of Windows and Linux, the stack trace has a simple and consistent form – consecutive stack frames of nested functions are chained together through stack frame pointers (saved values of the `EBP` register), with the current stack frame being pointed to by `EBP` itself. At any point of execution, it is possible to iterate through this chain to unwind the call stack and save the observed return addresses, as shown in Figure 9. This logic was implemented in our tool for 32-bit guest systems.

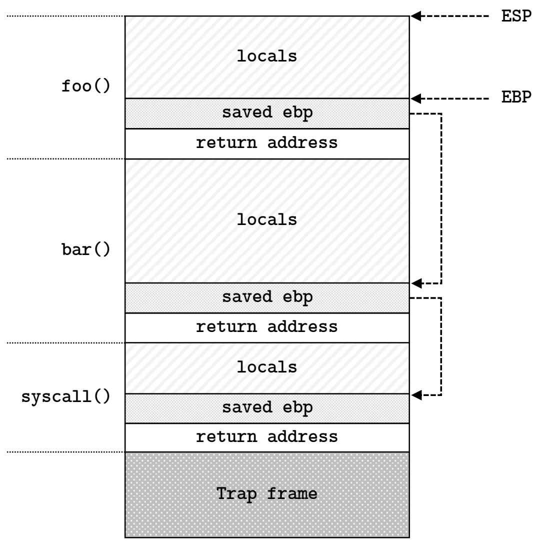

Figure 9: Example of a typical stack layout in x86 kernel code

In 64-bit Windows, the `RBP` register is not saved as part of the stack frame creation anymore, and thus the callstack cannot be traversed by the instrumentation without access to additional debug information. The necessary information is provided by Microsoft's debug symbols (`.pdb` files) corresponding to the kernel modules. Regardless of stack trace unwinding, our tool loads symbol files for every new detected driver for the purpose of address symbolization (see Section 3.2.3). In the presence of these debug symbols, obtaining the full callstack can be achieved using the `StackWalk64` API function [90] from the Debug Help Library (DbgHelp). As part of the input, the function expects to receive the full CPU context, which is available through the internal `BX_CPU_C` object. Another required primitive is a pointer to a custom `ReadMemoryRoutine` function, invoked by DbgHelp to read the virtual memory of the target process/kernel. In our case, it is a simple wrapper around `read_lin_mem` [72], a helper function for reading guest system memory. After putting these pieces together, retrieving the full call stack can be implemented as a straightforward loop over the `StackWalk64` call.

Support for 64-bit builds of Linux was not implemented, so we didn't study the problem in that configuration.

3.2.3 Address symbolization

The translation of raw addresses from the guest address space to meaningful symbols is essential for producing useful output reports. In this section, we describe how address symbolization was implemented on the Windows and Linux platforms.

Microsoft Windows The symbols for nearly all Windows system files are available on the Microsoft symbol server [85] and can be downloaded using the SymChk tool (`symchk.exe`), shipped with Debugging Tools for Windows. The command line syntax for downloading the symbols for a specific file to a chosen directory is as follows:

```
C:\> symchk.exe <driver> /s srv*<directory>*http://msdl.microsoft.com/
    download/symbols
```

Prior to starting each Bochspwn Reloaded session, we extracted all kernel modules from the guest filesystem, and downloaded their corresponding `.pdb` files to a single directory on the host machine. Later during Bochs run time, when our instrumentation discovered any new driver loaded in the emulated system, it would instantly look up symbols for that driver using the `SymLoadModule64` function [92]. Thanks to this, the Debug Help Library was always up to date with regards to the locations of kernel images in memory and their associated debug information. When the bug reports were generated, translating addresses into function names and offsets was achieved with a `SymFromAddr` [91] call, which performed the overall symbolization process and returned with the desired information.

Linux In the Linux-specific part of Bochspwn Reloaded, we didn't perform live address symbolization while the reports were produced by the tool. Instead, kernel addresses were printed in verbatim, and the log file was later subject to post-processing by a custom Python script. The script matched all kernel-mode addresses, symbolized them using the standard `addr2line` utility, and substituted them accordingly in the output file. This was possible by compiling the Linux kernel with ASLR disabled (`CONFIG_RANDOMIZE_BASE=n`), which guaranteed that the addresses in the instrumented system memory were compatible with the static `vmlinux` file on disk.

3.2.4 Breaking into the kernel debugger

The textual reports generated by our tool are verbose, but not always sufficient to understand the underlying vulnerabilities. One example of such scenario is when the leaked uninitialized data travels a long way (i.e. is copied across multiple locations in memory) before arriving at the final `memcpy` to user space. In cases like this, it is useful to attach a kernel debugger to the emulated system and learn about the memory layout and contents, kernel objects involved in the disclosure, the user-mode caller that invoked the affected system call, and any other important details about the state of the execution environment.

Both Windows and Linux support remote kernel debugging through COM ports. In the Bochs emulator on a Windows host, guest COM ports can be redirected to named Windows pipes by including the following line in the `bochsrc` configuration file:

```
com1: enabled=1, mode=pipe-server, dev=\\.\pipe\name
```

With the above configuration set up, it is possible to attach the WinDbg or gdb debuggers to the tested systems. However, this by itself is not enough to put the debuggers to good use, as long as we can't break the execution of the OS precisely at the moment of each new detected disclosure. To achieve this, we need assistance from the Bochs instrumentation.

In the x86(-64) architectures, breakpoints are installed by placing an `int3` instruction (opcode `0xcc`) at the desired location in code. This is no different in the emulated environment. To stop the kernel execution and pass control to the debugger, the instrumentation only needs to write the `0xcc` byte to `EIP` or `RIP`, depending on the system bitness. This suffices to break into the kernel debugger, but since the overwritten first byte of the next instruction is never restored in the above logic, the user has to do it themselves by looking up the value of the original byte in the executable image of the driver in question.

To address this inconvenience, our instrumentation declares a callback for the `bx_instr_interrupt` event invoked by Bochs every time an interrupt is generated, including the #BP trap triggered by the injected breakpoint. In that handler, we restore the previously saved value of the overwritten byte, thus bringing the original instruction back to its original form even before control is transferred to the kernel debugger. From the perspective of the emulated system, the #BP exception is generated for no apparent reason, as the `int3` instruction only lasts in memory for as long as it is needed to be fetched and executed by Bochs, and *disappears* shortly after.

Listing 19 presents an example WinDbg log from a brief investigation of a bug identified by Bochspwn Reloaded. The debugger informs us that a breakpoint exception was triggered, but the disassembly of memory under `RIP` shows the original `sub rdx, rcx` instruction, as further confirmed with the `u` command. To find out more about the circumstances of the leak, we check the value of the `RCX` register (the *destination* argument of `memmove`), to make sure that it points into user-mode memory. To follow up, we dump the memory area between `RDX` and `RDX+R8-1`, where `RDX` is the source address and `R8` is the number of bytes to copy. At offsets 0x4 through 0x7, we can observe the `0xaa` values, which is the filler byte for pool allocations. We can therefore assume that these bytes are the subject of the disclosure, and as we look closely at the contents of the buffer, we can also deduce that it is a `UNICODE_STRING` structure followed by the corresponding textual data. Lastly, we display the stack trace to establish how the control flow reached the current point. After our analysis is completed, we can continue the system execution with the `g` command.

3.2.5 Address space visualization

The taint information stored by Bochspwn Reloaded at any given time of the guest system run time may be used to display the layout of the kernel address space as an image. Graphical representation provides valuable insight into the inner workings of the dynamic allocator and memory manager. It also makes it possible to observe memory consumption under various conditions, as well as compare the behavior of different versions of an operating system. Finally, it also proves useful for security research – for example, memory visualization may help in fine-tuning pool spraying techniques, or calibrating a pool-massaging algorithm designed to facilitate the exploitation of use-after-free vulnerabilities.

Several projects for visualizing process and kernel address spaces have been devised so far. MemSpyy [99] and MemoryDisplay [106] display maps of the virtual memory of Windows userland applications. KernelMAP [67] displays the locations of drivers, objects, locks and CPU structures in the Windows kernel memory. An improved version of the tool called MemMAP [15] also shows kernel stacks and GDI objects, as well as memory maps of ring 3 programs. Radocea and Wicherski further demonstrated successful visualization of page tables on Android, OS X and iOS [20]. The main advantage of performing the visualization in an x86 emulator is that the address space can be observed at very early boot stages, which would be otherwise impossible to achieve from within the guest system itself.

As an exercise, we implemented the feature in Bochspwn Reloaded for 32-bit guests. Every N seconds, a separate thread in the emulator would make a snapshot of the current state of the shadow memory, marking each of the existing memory pages[5] as *free*, *stack* or *heap*. The snapshots were then converted to bitmaps of a 1024x{256,512} resolution, where each pixel represented a single page. Depending on their type, the pixels were colored black (unused), green (stack) or red (heap/pool). Figure 10 shows the Windows 7 kernel memory layout after 40 minutes of booting up and running several initial ReactOS tests. Figure 11 shows the layout on Windows 10 after 120 minutes of run time. Finally, Figure 12 shows the memory state of Ubuntu 16.04 after 60 minutes of run time, which included booting up, running the Trinity fuzzer [13] and starting a few initial tests from the Linux Test Project [8].

[5]There are `0x80000` (524,288) kernel pages on Windows, and `0x40000` (262,144) kernel pages on Linux in the 3G/1G memory split configuration.

```
kd> g
Break instruction exception - code 80000003 (first chance)
nt!memmove+0x3:
fffff800`026fc603 482bd1          sub     rdx,rcx

kd> u
nt!memmove+0x3:
fffff800`026fc603 482bd1          sub     rdx,rcx
fffff800`026fc606 0f829e010000    jb      nt!memmove+0x1aa
fffff800`026fc60c 4983f808        cmp     r8,8
fffff800`026fc610 7262            jb      nt!memmove+0x74
fffff800`026fc612 f6c107          test    cl,7
fffff800`026fc615 7437            je      nt!memmove+0x4e
fffff800`026fc617 f6c101          test    cl,1
fffff800`026fc61a 740c            je      nt!memmove+0x28

kd> ? rcx
Evaluate expression: 2554392 = 00000000`0026fa18

kd> db /c 8 rdx rdx+r8-1
fffff8a0`00ff1d20  2e 00 30 00 aa aa aa aa  ..0.....
fffff8a0`00ff1d28  30 1d ff 00 a0 f8 ff ff  0.......
fffff8a0`00ff1d30  5c 00 44 00 65 00 76 00  \.D.e.v.
fffff8a0`00ff1d38  69 00 63 00 65 00 5c 00  i.c.e.\.
fffff8a0`00ff1d40  48 00 61 00 72 00 64 00  H.a.r.d.
fffff8a0`00ff1d48  64 00 69 00 73 00 6b 00  d.i.s.k.
fffff8a0`00ff1d50  56 00 6f 00 6c 00 75 00  V.o.l.u.
fffff8a0`00ff1d58  6d 00 65 00 32 00 00 00  m.e.2...

kd> k
 # Child-SP          RetAddr           Call Site
00 fffff880`03d0b8c8 fffff800`02a75319 nt!memmove+0x3
01 fffff880`03d0b8d0 fffff800`02938426 nt!IopQueryNameInternal+0x289
02 fffff880`03d0b970 fffff800`0294cfa8 nt!IopQueryName+0x26
03 fffff880`03d0b9c0 fffff800`0297713b nt!ObpQueryNameString+0xb0
04 fffff880`03d0bac0 fffff800`0271d283 nt!NtQueryVirtualMemory+0x5fb
05 fffff880`03d0bbb0 00000000`77589ada nt!KiSystemServiceCopyEnd+0x13
[...]

kd> g
```

Listing 19: WinDbg log after an infoleak in `nt!IopQueryNameInternal` is flagged by Bochspwn Reloaded (CVE-2018-0894)

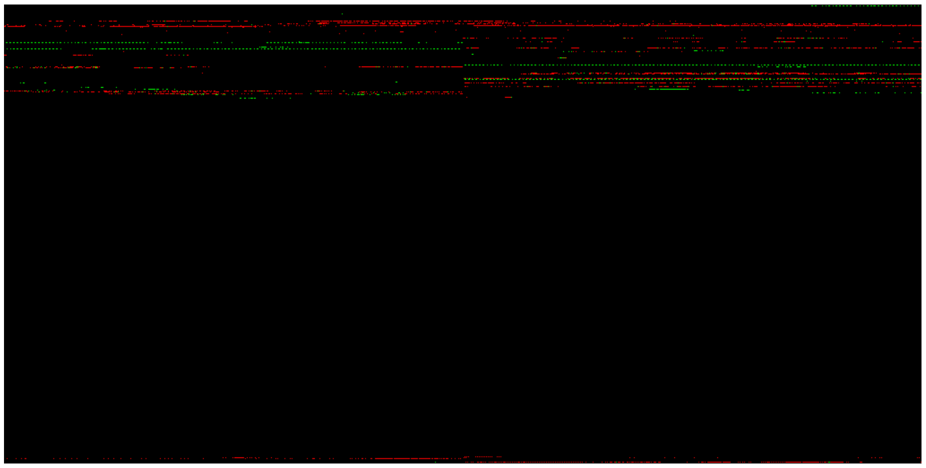

Figure 10: Windows 7 kernel address space layout; green: stack pages, red: pool pages

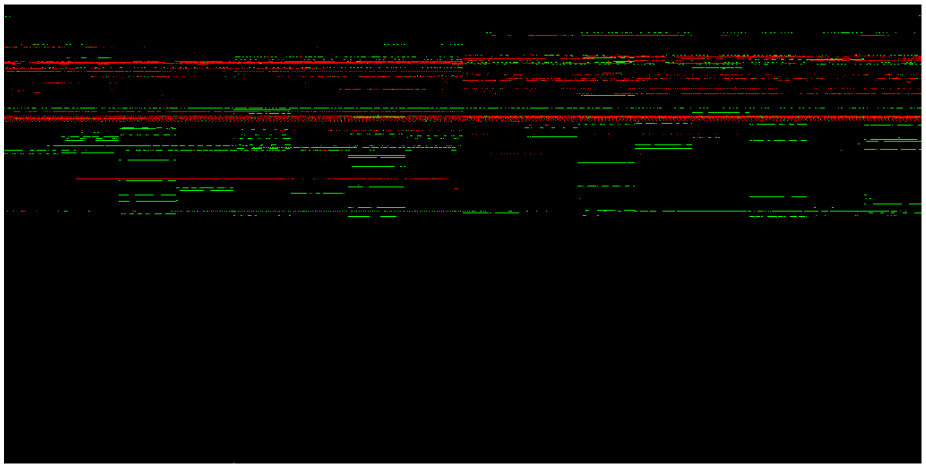

Figure 11: Windows 10 kernel address space layout; green: stack pages, red: pool pages

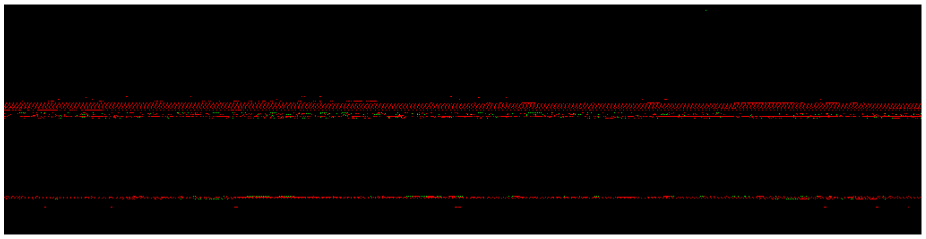

Figure 12: Ubuntu 16.04 kernel address space layout; green: stack pages, red: heap pages

3.3 Performance

In this section, we examine the general performance of the instrumentation by comparing its CPU and memory usage to the same guest operating systems run in a regular virtual machine (VirtualBox) and a non-instrumented Bochs emulator. The aim of this section is to provide a general overview of the overhead associated with the proposed approach; the numbers presented here are approximate and should not be considered as accurate benchmarks.

The testing was performed on a workstation equipped with an Intel Xeon E5-1650 v4 @ 3.60 GHz CPU, 64 GB of DDR4-2400 RAM, and a Samsung 850 PRO 512 GB SSD (SATA 3); the guest systems were assigned 1 processor core and 2 GB of physical memory.

3.3.1 CPU overhead

The boot times of the systems tested as part of this research in three different setups are shown in Table 2. *Boot time* is defined as the time period from cold boot to a fully loaded log-on screen. The data was obtained by testing each configuration in several (typically three) attempts and accepting the lowest value, in hope to minimize the influence of the host machine's background load on the results.

Unsurprisingly, it is visible that operating systems run significantly slower in a software emulator compared to a virtual machine; in our case, the slowdown factor was observed to be around 13-18X. Under such non-instrumented emulators, the guests usually ran at a rate of 80-150 million IPS (*Instructions Per Second*) with reasonably responsive graphical interfaces. As the boot times all fall within a 10-minute mark, we consider the execution speed provided by Bochs to be sufficient for research purposes.

In order to keep the inevitable overhead modest, it is essential for any additional instrumentation to be implemented as efficiently as possible, especially when using frequently invoked callbacks such as `bx_instr_before_execution` or `bx_instr_lin_access`. In Bochspwn Reloaded, the average emulation speed drops to around 30-50 MIPS, which seems to correctly correlate with the increase in measured boot times of around 2.2 to 2.65X compared to unmodified Bochs. Even with the aggregate slowdown multiplier between 32-50X in relation to a regular VM, all tested environments remained responsive, and the test cases and programs designed to extend the kernel code coverage successfully completed within 24 hours of booting the emulated systems up.

	Windows 7		Windows 10		Ubuntu 16.10
	x86	x64	x86	x64	x86
VirtualBox	00:07	00:11	00:24	00:22	00:25
Bochs	01:33	02:20	05:47	06:52	06:38
Bochspwn Reloaded	03:45	05:55	12:43	18:04	17:35

Table 2: Time from cold boot to log-on screen in tested configurations (mm:ss)

	Windows 7		Windows 10		Ubuntu 16.10
	x86	x64	x86	x64	x86
Bochs	2126				
Bochspwn Reloaded	8300	4220	8290	6628	5205

Table 3: Peak memory consumption of the `bochs.exe` process between cold boot and log-on screen of the guest systems (in MB)

3.3.2 Memory overhead

The peak memory usage of the `bochs.exe` emulator process for both non-instrumented and instrumented Bochs is shown in Table 3, measured as the "Peak Private Bytes" value shown by Process Explorer [10] when the log-on screen was loaded by the guest OS. In the non-instrumented case, memory consumption was fixed at around 2126 MB for all tested systems, the bulk of which corresponds to the 2 GB of emulated physical memory assigned to the guests. Any memory allocated beyond that point can be accounted as an overhead of the Bochspwn Reloaded project.

In the 32-bit versions of our instrumentation, the shadow memory and other auxilliary metadata are allocated statically and induce a constant overhead of 6 GB for Windows and 3 GB for Linux, as explained in Section 3.1.1. This is correctly reflected in the observed data. On the contrary, the memory consumption is variable in case of emulated x64 platforms, where both the taint and allocation origin descriptors are allocated on demand. For Windows 7 and Windows 10 64-bit, the memory overhead at the time of the log-on screen was 2094 MB and 4502 MB, respectively. Somewhat unintuitively, the presented virtual memory usage is higher for the x86 architecture than x64, but it should be noted that (a) the overhead on x64 could grow beyond the 6 GB boundary during system run time, while it remains constant on x86, and (b) accesses to metadata on x64 come with extra computational costs related to the custom memory overcommittment mechanism and the overhead of hash map operations.

In summary, the memory requirements of the project are significant, but can be met by any modern workstation with at least 16 GB of RAM installed.

3.4 Testing

The effectiveness of any instrumentation-based vulnerability detection is as good as the code coverage achieved against the tested software. With this in mind, we attempted to maximize the coverage of the analyzed kernels using publicly available tools and methodology, while running them inside our tool. In the subsections below, we explain how the testing was carried out on the Windows and Linux platforms.

3.4.1 Microsoft Windows

As part of the research project, we ran Bochspwn Reloaded against Windows 7 and Windows 10, both 32 and 64-bit builds. The middle version of the operating system – Windows 8.1 – was excluded from the analysis. The assumption behind this approach was that the testing of Windows 7 would reveal bugs that had been internally fixed by Microsoft in newer systems but not backported to the older ones, while instrumentation of Windows 10 would uncover bugs in the most recently introduced kernel code. In this context, there was very little attack surface in Windows 8.1 that wouldn't be already covered by the testing of the two other versions, and as such, we considered it redundant to analyze all three major releases of the OS.

In terms of bitness, in our experience and by intuition, a majority of kernel infoleaks are cross-platform and affect both x86 and x64 builds of the code. This is related to the fact that most root causes of the bugs, such as explicitly uninitialized variables, structure fields, arrays etc. are bitness-agnostic and reproduce in both execution modes. The subset of issues limited to x86 is very narrow, as there are no fundamental reasons for the presence of such 32-bit only disclosures other than low level platform-specific code (e.g. exception handling). On the other hand, x64-only bugs may exist due to the fact that the width of certain data types (`size_t`, pointers etc.) and their corresponding alignment requirements increase from 4 to 8 bytes, which in turn:

- creates new alignment holes in structures; one example being the standard `UNICODE_STRING` structure,
- extends the size of unions, which may misalign their fields and introduce new uninitialized bytes, as is the case in `IO_STATUS_BLOCK`.

Throughout most of 2017, we developed instrumentation for 32-bit guest systems, which yielded a total of 48 CVEs assigned by Microsoft. At the end of that year, we implemented support for 64-bit platforms, which uncovered further 17 x64-specific issues. The results of the project are detailed in Section 3.5.

The testing process of Windows in the emulated environment involved the following steps aimed to maximize the kernel code coverage:

- Booting up the system.
- Starting and navigating through a number of default programs, accessories and administrative tools, including File Explorer, Internet Explorer, Calculator, Notepad, WordPad, Command Line, Registry Editor, Paint, Windows Media Player, XPS Viewer, Control Panel and so forth.
- Running more than 800 *winetests* and *rostests* [2] that are part of the ReactOS project. They are API tests designed to check that the ReactOS implementations behave in the same way that Microsoft APIs do. Due to the fact that they invoke a variety of system interfaces and test them extensively, they are the perfect means to extend the kernel code coverage.

- Running and navigating through the code samples [32] for the "Windows Graphics Programming: Win32 GDI and DirectDraw" book [31].
- Running around 30 `NtQuery` test suites developed specifically to uncover memory disclosure in the specific subset of the system calls for querying information about objects in the system. These test programs are further discussed in Section 5.4 "Differential syscall fuzzing".
- Shutting down the system.

As shown, Windows was tested on a best effort basis, and there is much room for improvement in terms of coupling existing dynamic binary instrumentation schemes with new ways of exploring a more substantial portion of the kernel code. However, we believe that the steps we took allowed us to identify most of the easily discoverable bugs that other parties could likely run across.

3.4.2 Linux

The Linux platform subject to examination was Ubuntu Server 16.10 32-bit with a custom-compiled kernel v4.8. Support for x64 builds of the kernel was never implemented. As part of the testing, we executed the following actions in the system:

- Booting up the system.
- Logging in via SSH.
- Running several standard command-line utilities operating on processes, file system and the network.
- Running the full set of the Linux Test Project (LTP) [8] unit tests.
- Running the iknowthis [105] syscall fuzzer for a day.
- Running the Trinity [13] syscall fuzzer for a day.
- Shutting down the system.

Currently, the state-of-the-art Linux syscall fuzzer is syzkaller [11]. It is compatible with the various *Sanitizers* (most importantly KASAN [12]), uses code coverage information to guide the fuzzing, and has identified dozens of bugs in the Linux kernel to date [5]. Unfortunately, syzkaller couldn't be run inside Bochspwn Reloaded, as it only supported the x64 and arm64 architectures while our tool was limited to x86 builds of the system.

3.5 Results

In this section, we present a summary of previously unknown vulnerabilities discovered by running Bochspwn Reloaded against Windows and Linux.

3.5.1 Microsoft Windows

Throughout the development of the project in 2017 and early 2018, we ran multiple iterations of the instrumentation on the then-latest builds of Windows 7 and 10. All issues found in each session were promptly reported to Microsoft in accordance with the Google Project Zero disclosure policy. The first identified vulnerability was reported on March 1, 2017 [59], and the last one was sent to the vendor on January 22, 2018 [53]. In that time period, we progressively improved the tool and introduced new features, which enabled us to regularly uncover new layers of infoleaks. This is reflected in the history of the bug reports – for example, the first 12 reported issues were pool-based disclosures, because the handling of stack allocations was added a few weeks later. Similarly, we initially developed instrumentation for 32-bit guest systems, and only implemented support for 64-bit platforms at the end of 2017. This explains why all of the x64-specific leaks we discovered were patched between February and April 2018.

In total, we filed 73 issues describing Windows kernel memory disclosure to user-mode in the Project Zero bug tracker. Out of those, 69 were closed as "Fixed", 2 as "Duplicate" and 2 as "WontFix". The duplicate reports were caused by the fact that some leaks reported as separate issues were determined by the vendor to be caused by a single vulnerability in the code. The WontFix cases were valid bugs, but turned out to be only reachable from a privileged `dwm.exe` process and hence didn't meet the bar to be serviced in a security bulletin.

Microsoft classified the problems as 65 unique security flaws. The discrepancy between the number of CVEs and "Fixed" bug tracker entries stems from the fact that the vendor combined several groups of issues into one, e.g. if they considered the bugs to have a common logic root cause. In cases where this was inconsistent with our assessment and we viewed them as separate bugs, we marked the corresponding tracker issues as "Fixed" instead of "Duplicate".

Two example reports generated for the CVE-2017-8473 and CVE-2018-0894 vulnerabilities are shown in Listings 20 and 21. A complete summary of memory disclosure vulnerabilities found by the tool is presented in Tables 4 and 5, while Figure 13 illustrates the distribution of disclosed memory types. We expect that stack leaks are more prevalent than pool leaks due to the fact that a majority of temporary objects constructed by the kernel in system call handlers are allocated locally. Furthermore, Figure 14 shows a classification of the bugs based on the kernel modules they were found in. The core `ntoskrnl.exe` executable image was the most suspectible to infoleaks, likely due to its large attack surface and our intensified testing of the `NtQuery` syscall family. The graphical `win32k.sys` driver was also affected by a significant number of problems, both in regular syscall handlers and a mechanism known as *user-mode callbacks*. Lastly, several individual issues were found in other drivers such as `partmgr.sys`, `mountmgr.sys` or `nsiproxy.sys`, mostly in the handling of user-accessible IOCTLs with complex output structures.

```
------------------------------ found uninit-access of address 94447d04
[pid/tid: 000006f0/00000740] {    explorer.exe}
       READ of 94447d04 (4 bytes, kernel--->user), pc = 902df30f
       [ rep movsd dword ptr es:[edi], dword ptr ds:[esi] ]

[Pool allocation not recognized]
Allocation origin: 0x90334988 (win32k.sys!__SEH_prolog4+00000018)

Destination address: 1b9d380
Shadow bytes: 00 ff ff ff Guest bytes: 00 bb bb bb

Stack trace:
 #0  0x902df30f (win32k.sys!NtGdiGetRealizationInfo+0000005e)
 #1  0x8288cdb6 (ntoskrnl.exe!KiSystemServicePostCall+00000000)
```

Listing 20: Report of the CVE-2017-8473 bug detected on Windows 7 32-bit

```
------------------------------ found uninit-copy of address fffff8a000a63010

[pid/tid: 000001a0/000001a4] {     wininit.exe}
       COPY of fffff8a000a63010 ---> 1afab8 (64 bytes), pc = fffff80002698600
       [                              mov r11, rcx ]
Allocation origin: 0xfffff80002a11101
                   (ntoskrnl.exe!IopQueryNameInternal+00000071)
--- Shadow memory:
00000000: 00 00 00 00 ff ff ff ff 00 00 00 00 00 00 00 00 ................
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
--- Actual memory:
00000000: 2e 00 30 00 aa aa aa aa 20 30 a6 00 a0 f8 ff ff ..0..... 0......
00000010: 5c 00 44 00 65 00 76 00 69 00 63 00 65 00 5c 00 \.D.e.v.i.c.e.\.
00000020: 48 00 61 00 72 00 64 00 64 00 69 00 73 00 6b 00 H.a.r.d.d.i.s.k.
00000030: 56 00 6f 00 6c 00 75 00 6d 00 65 00 32 00 00 00 V.o.l.u.m.e.2...
--- Stack trace:
 #0  0xfffff80002698600 (ntoskrnl.exe!memmove+00000000)
 #1  0xfffff80002a11319 (ntoskrnl.exe!IopQueryNameInternal+00000289)
 #2  0xfffff800028d4426 (ntoskrnl.exe!IopQueryName+00000026)
 #3  0xfffff800028e8fa8 (ntoskrnl.exe!ObpQueryNameString+000000b0)
 #4  0xfffff8000291313b (ntoskrnl.exe!NtQueryVirtualMemory+000005fb)
 #5  0xfffff800026b9283 (ntoskrnl.exe!KiSystemServiceCopyEnd+00000013)
```

Listing 21: Report of the CVE-2018-0894 bug detected on Windows 7 64-bit

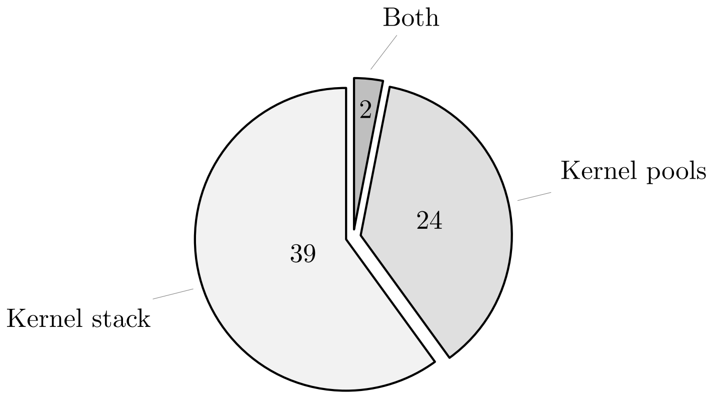

Figure 13: Distribution of disclosed memory types between flaws discovered in Windows

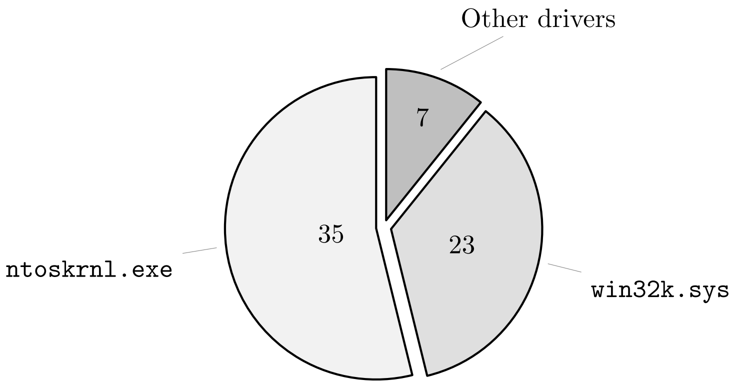

Figure 14: Distribution of affected kernel modules between flaws discovered in Windows

CVE ID	Component	Fix Date	Leaked bytes	x64 only
CVE-2017-0258	`ntoskrnl.exe`	May 2017	8	
CVE-2017-0259	`ntoskrnl.exe`	May 2017	60	
CVE-2017-8462	`ntoskrnl.exe`	June 2017	1	
CVE-2017-8469	`partmgr.sys`	June 2017	484	
CVE-2017-8484	`win32k.sys`	June 2017	5	
CVE-2017-8488	`mountmgr.sys`	June 2017	14	
CVE-2017-8489[6]	`ntoskrnl.exe`	June 2017	6 or 72	
CVE-2017-8490	`win32k.sys`	June 2017	6672	
CVE-2017-8491	`volmgr.sys`	June 2017	8	
CVE-2017-8492	`partmgr.sys`	June 2017	4	
CVE-2017-8564	`nsiproxy.sys`	July 2017	13	
CVE-2017-0299[7]	`ntoskrnl.exe`	August 2017	2	
CVE-2017-8680	`win32k.sys`	September 2017	Arbitrary	
CVE-2017-11784	`ntoskrnl.exe`	October 2017	192	
CVE-2017-11785	`ntoskrnl.exe`	October 2017	56	
CVE-2017-11831	`ntoskrnl.exe`	November 2017	25	
CVE-2018-0746	`ntoskrnl.exe`	January 2018	12	
CVE-2018-0810[8]	`win32k.sys`	February 2018	4	✓
CVE-2018-0813	`win32k.sys`	March 2018	4	✓
CVE-2018-0894	`ntoskrnl.exe`	March 2018	4	✓
CVE-2018-0898	`ntoskrnl.exe`	March 2018	8	✓
CVE-2018-0899	`videoprt.sys`	March 2018	20	✓
CVE-2018-0900	`ntoskrnl.exe`	March 2018	40	✓
CVE-2018-0926	`win32k.sys`	March 2018	4	✓
CVE-2018-0972	`ntoskrnl.exe`	April 2018	8	
CVE-2018-0973[9]	`ntoskrnl.exe`	April 2018	4	

Table 4: A summary of discovered Windows pool memory disclosure bugs

[6]The CVE was assigned to a generic mitigation of zeroing the Buffered I/O output buffer [39]. It fixed two bugs in IOCTLs handled by the `\Device\KsecDD` and `\\.\WMIDataDevice` devices, filed in the Google Project Zero bug tracker as issues 1147 and 1152.

[7]A patch for the vulnerability first shipped in June 2017. After it was proven ineffective, Microsoft released a revised version of the fix in August of the same year.

[8]The CVE collectively covers four different memory disclosure bugs found in `win32k.sys` user-mode callbacks – one pool-based and three stack-based leaks. They were filed in the Google Project Zero bug tracker as issues 1467, 1468, 1485 and 1487.

[9]The vulnerability disclosed uninitialized pool memory on Windows 7, and stack memory on Windows 10.

CVE ID	Component	Fix Date	Leaked bytes	x64 only
CVE-2017-0167	`win32k.sys`	April 2017	20	
CVE-2017-0245	`win32k.sys`	May 2017	4	
CVE-2017-0300	`ntoskrnl.exe`	June 2017	5	
CVE-2017-8470	`win32k.sys`	June 2017	50	
CVE-2017-8471	`win32k.sys`	June 2017	4	
CVE-2017-8472	`win32k.sys`	June 2017	7	
CVE-2017-8473	`win32k.sys`	June 2017	8	
CVE-2017-8474	`ntoskrnl.exe`	June 2017	8	
CVE-2017-8475	`win32k.sys`	June 2017	20	
CVE-2017-8476	`ntoskrnl.exe`	June 2017	4	
CVE-2017-8477	`win32k.sys`	June 2017	104	
CVE-2017-8478	`ntoskrnl.exe`	June 2017	4	
CVE-2017-8479	`ntoskrnl.exe`	June 2017	16	
CVE-2017-8480	`ntoskrnl.exe`	June 2017	6	
CVE-2017-8481	`ntoskrnl.exe`	June 2017	2	
CVE-2017-8482	`ntoskrnl.exe`	June 2017	32	
CVE-2017-8485	`ntoskrnl.exe`	June 2017	8	
CVE-2017-8677	`win32k.sys`	September 2017	8	
CVE-2017-8678	`win32k.sys`	September 2017	4	
CVE-2017-8681	`win32k.sys`	September 2017	128	
CVE-2017-8684	`win32k.sys`	September 2017	88	
CVE-2017-8685	`win32k.sys`	September 2017	1024	
CVE-2017-8687	`win32k.sys`	September 2017	8	
CVE-2017-11853	`win32k.sys`	November 2017	12	
CVE-2018-0745	`ntoskrnl.exe`	January 2018	4	
CVE-2018-0747	`ntoskrnl.exe`	January 2018	4	
CVE-2018-0810	`win32k.sys`	February 2018	4 or 8	
CVE-2018-0832	`ntoskrnl.exe`	February 2018	4	
CVE-2018-0811	`win32k.sys`	March 2018	4	✓
CVE-2018-0814	`win32k.sys`	March 2018	8	✓
CVE-2018-0895	`ntoskrnl.exe`	March 2018	4	✓
CVE-2018-0896	`msrpc.sys`	March 2018	8	✓
CVE-2018-0897	`ntoskrnl.exe`	March 2018	120	✓
CVE-2018-0901	`ntoskrnl.exe`	March 2018	4	✓
CVE-2018-0968	`ntoskrnl.exe`	April 2018	4	✓
CVE-2018-0969	`ntoskrnl.exe`	April 2018	4	
CVE-2018-0970	`ntoskrnl.exe`	April 2018	4 or 16	
CVE-2018-0971	`ntoskrnl.exe`	April 2018	4	✓
CVE-2018-0973	`ntoskrnl.exe`	April 2018	4	✓
CVE-2018-0974	`ntoskrnl.exe`	April 2018	8	✓
CVE-2018-0975	`ntoskrnl.exe`	April 2018	4 or 56	

Table 5: A summary of discovered Windows stack memory disclosure bugs

```
------------------------------ found uninit-access of address f5733f38
========== READ of f5733f38 (4 bytes, kernel--->kernel), pc = f8aaf5c5
                           [          mov edi, dword ptr ds:[ebx+84] ]
[Heap allocation not recognized]
Allocation origin: 0xc16b40bc: SYSC_connect at net/socket.c:1524
Shadow bytes: ff ff ff ff Guest bytes: bb bb bb bb
Stack trace:
#0 0xf8aaf5c5: llcp_sock_connect at net/nfc/llcp_sock.c:668
#1 0xc16b4141: SYSC_connect at net/socket.c:1536
#2 0xc16b4b26: SyS_connect at net/socket.c:1517
#3 0xc100375d: do_syscall_32_irqs_on at arch/x86/entry/common.c:330
  (inlined by) do_fast_syscall_32 at arch/x86/entry/common.c:392
```

Listing 22: Report of a bug in `llcp_sock_connect` on Ubuntu 16.10 32-bit

3.5.2 Linux

We implemented support for 32-bit Linux kernels in April 2017, including the same infoleak detection logic that had been successfully used for Windows. As a result of instrumenting Ubuntu 16.10 for several days, we detected a single minor bug – disclosure of 7 uninitialized kernel stack bytes in the processing of specific IOCTLs in the `ctl_ioctl` function (`drivers/md/dm-ioctl.c`). The routine handles requests sent to the `/dev/control/mapper` device, which is only accessible by the `root` user, significantly reducing the severity of the issue. We identified the problem on April 20, but before we were able to submit a patch, we learned that it had been independently fixed by Adrian Salido in commit `4617f564c0` [16] on April 27.

The lack of success in identifying new infoleaks in Linux can be explained by the vast extent of work done to secure the kernel in the past. Accordingly, we decided to extend the detection logic to include all references to uninitialized memory. By making this change, we intended to uncover other, possibly less dangerous bugs, where uninitialized memory was used in a meaningful way, but not directly copied to the user space. Thanks to this approach, we discovered further 15 bugs – one disclosure of uninitialized stack memory through `AF_NFC` sockets (see the corresponding report presented in Listing 22) and 14 lesser, functional issues in various subsystems of the kernel, with limited security impact. The combined results of this effort are enumerated in Table 6.

During and after the triage of the output reports, we noticed that some of our findings collided with the work of independent researchers and developers; i.e. several bugs had been fixed days or weeks prior to our discovery, or reported by other parties shortly after we submitted patches. Most collisions occurred with the KernelMemorySanitizer project [6], which was actively developed and used to test Linux in the same period. Consequently, we only submitted 11 patches for the 16 uncovered bugs, as the rest had already been addressed. This example is very illustrative of the velocity of improvements applied to Linux, and the scope of work done to eliminate entire vulnerability classes.

File	Fix commit	Collision	Mem.	Type
`net/nfc/llcp_sock.c`	`608c4adfca`	✓	Stack	L
`drivers/md/dm-ioctl.c`	`4617f564c0`	✓	Stack	L
`net/bluetooth/l2cap_sock.c` `net/bluetooth/rfcomm/sock.c` `net/bluetooth/sco.c`	`d2ecfa765d`		Stack	U
`net/caif/caif_socket.c`	`20a3d5bf5e`		Stack	U
`net/iucv/af_iucv.c`	`e3c42b61ff`		Stack	U
`net/nfc/llcp_sock.c`	`f6a5885fc4`		Stack	U
`net/unix/af_unix.c`	`defbcf2dec`		Stack	U
`kernel/sysctl_binary.c`	`9380fa60b1`	✓	Stack	U
`fs/eventpoll.c`	`c857ab640c`	✓	Stack	U
`kernel/printk/printk.c`	`5aa068ea40`	✓	Heap	U
`net/decnet/netfilter/dn_rtmsg.c`	`dd0da17b20`		Heap	U
`net/netfilter/nfnetlink.c`	`f55ce7b024`		Heap	U
`fs/ext4/inode.c`	`2a527d6858`	✓	Stack	U
`net/ipv4/fib_frontend.c`	`c64c0b3cac`	✓	Heap	U
`fs/fuse/file.c`	`68227c03cb`		Heap	U
`arch/x86/kernel/alternative.c`	`fc152d22d6`		Stack	U

Table 6: A summary of discovered uninitialized memory bugs in Linux.
`L`: kernel memory disclosure; `U`: use of uninitialized memory.

4 Windows bug reproduction techniques

Bug reproduction is an integral part of hunting for memory disclosure vulnerabilities, as it (a) helps make sure that a problem identified by Bochspwn Reloaded is an actual bug in the kernel and not in the instrumentation, and (b) helps vendors clearly observe the problem in the same way we see it in our test environment. For these reasons, we have developed a unified set of system configuration steps and methods of writing our proof-of-concept programs to achieve the most deterministic behavior possible. In the paragraphs below, we discuss our approaches to reproducing Windows pool-based and stack-based disclosures. In both cases, we used a VirtualBox VM with the desired guest operating system, typically Windows 7 or 10, 32-bit or 64-bit.

In a number of cases, attempting reproduction was preceded by long hours spent on analyzing Bochspwn Reloaded reports and the internal state of the system in the kernel debugger, in order to fully understand the root cause of the bugs first.

Kernel pools Reproducing pool-based leaks in Windows is a relatively easy task. The system supports a special option in the kernel called *Special Pool* [79], which helps detect pool buffer overflows and underflows by allocating each memory chunk at the beginning or end of a separate page. It can be controlled with a default program called *Driver Verifier Manager* (`verifier.exe`).

One useful property of the mechanism is that it fills the body of every new allocation with a unique, repeated marker byte. If leaked to user-mode, these variable marker bytes can be seen at common offsets of the PoC program output. The only requirement is to determine which kernel module requests the affected allocation (which is usually answered by Bochspwn Reloaded) and enable special pool for that module. Then, we should see output similar to the following after starting the proof-of-concept twice (on the example of CVE-2017-8491):

```
D:\>VolumeDiskExtents.exe
00000000: 01 00 00 00 39 39 39 39 ....9999
00000008: 00 00 00 00 39 39 39 39 ....9999
00000010: 00 00 50 06 00 00 00 00 ..P.....
00000018: 00 00 a0 f9 09 00 00 00 ........

D:\>VolumeDiskExtents.exe
00000000: 01 00 00 00 2f 2f 2f 2f ....////
00000008: 00 00 00 00 2f 2f 2f 2f ....////
00000010: 00 00 50 06 00 00 00 00 ..P.....
00000018: 00 00 a0 f9 09 00 00 00 ........
```

It is worth noting that special pool is only applied to allocations of up to around 4080 bytes, which should however not be a significant restriction for practical purposes. As the size of the special pool increases with the amount of RAM, we recommend assigning the test VM with enough physical memory to ensure that all special pool requests can be fulfilled.

Kernel stack Reliable reproduction of kernel stack disclosure is more troublesome than leaks from the pools, as there is no official or documented way to fill kernel stack frames with marker bytes at the time of allocation. These markers are essential, because otherwise the leaked bytes contain non-deterministic leftover memory which is often either uninteresting or plainly filled with zeros. This makes it difficult to distinguish legitimate syscall output from uninitialized data, let alone hope that the proof-of-concept output observed by the vendor will be similar to ours.

In order to address this problem, we decided to use a *kernel stack spraying* technique [50, 46]. As mentioned in Section 2.2.3, many system call handlers in Windows load small chunks of input data into stack-based buffers, and only request pool allocations when larger memory blocks are required. This optimization reduces the number of allocator invocations by using the capacity of the stack. As it turns out, the meaning of *small* and *large* is flexible in the Windows kernel, with some syscalls using local helper buffers of up to several kilobytes in size before turning to pool allocations. This allows user-mode programs to fill a vast portion of the kernel stack with fully controlled data, which proves very useful for demonstrating memory disclosure vulnerabilities.

Candidates of syscalls capable of spraying the kernel stack can be identified by looking for functions with large stack frames whose names start with “Nt”. In our reproducers, we used the following services:

- `nt!NtMapUserPhysicalPages` – Sprays up to 4096 bytes on 32-bit systems and 8192 bytes on 64-bit systems.
- `win32k!NtGdiEngCreatePalette` – Sprays up to 1024 bytes of the kernel stack, and may return a new GDI palette object which needs to be destroyed later on.

Both system calls can be invoked through their corresponding wrappers in the `ntdll.dll` and `gdi32.dll` libraries, which is useful for cross-version system compatibility. The overall process of using kernel stack spraying to achieve reliable reproduction is illustrated in Figure 15. Upon implementing the idea, it should be simple to spot the markers in the syscall output where kernel data is leaking, as shown on the example of CVE-2017-8473:

```
D:\>NtGdiGetRealizationInfo.exe
00000000: 10 00 00 00 03 01 00 00 ........
00000008: 2e 00 00 00 69 00 00 46 ....i..F
00000010: 41 41 41 41 41 41 41 41 AAAAAAAA
```

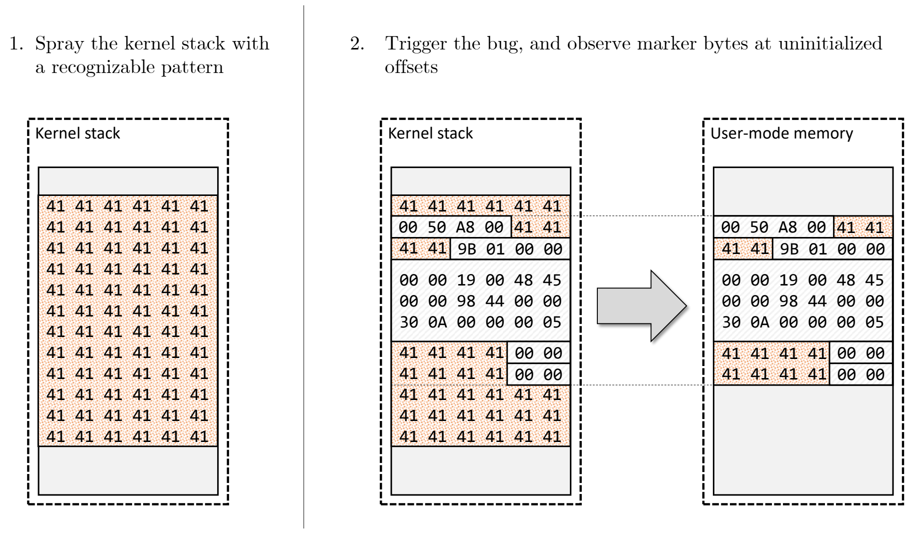

Figure 15: Kernel stack spraying used for memory disclosure reproduction

5 Alternative detection methods

Kernel memory disclosure can be detected with a variety of techniques, which complement each other in how they approach the problem. Full system emulation is one of them, and while effective, it is limited by the reachable kernel code coverage. In the following sections, we discuss alternative methods of identifying infoleaks, outlining their feasibility, pros and cons, and the results of evaluating some of them against the Windows kernel.

5.1 Static analysis

Static analysis was shown to be successful in finding Linux kernel bugs in the past, both by analyzing the source code for simple, known-vulnerable patterns [97, 98], and by running full-fledged memory tainting during compilation to identify execution paths which may lead to memory disclosure [45]. The prime advantage of the method over dynamic techniques is the fact that it is capable of processing entire code bases without the need to actually execute them. On the other hand, depending on the design decisions and implementation, they may yield multiple false positives and/or false negatives. For example, the UniSan authors admitted [45] that due to limited time resources and a conservative approach taken by their project, they were only able to manually analyze about 20% of all allocations marked as potentially leaking uninitialized memory. Overall, we consider static analysis to be a powerful technique that complements other dynamic methods well.

Given the closed-source nature of Windows, existing tools devised for Linux cannot be tested against Microsoft's operating system by anyone other than the vendor itself. It is difficult to estimate how much work is done internally in this area by Microsoft due to limited public documentation of such efforts. However, we are hoping that Microsoft will continue to invest in static analysis and will adopt the current state-of-the-art techniques, as they are in the best position to design and run advanced source-code analysis on the Windows kernel.

Binary-level static analysis is another viable option, but much more difficult to implement effectively, as a significant portion of useful information – such as structure definitions and variable types – is stripped away during compilation. For example, we expect that Peiró's model checking [97] could be ported to Windows with relative ease, to search for stack-based objects which are only partially initialized in their parent function and subsequently copied to user-mode. However, we haven't tested this method in practice.

5.2 Manual code review

Earlier in the paper, we stated that memory disclosure in software written in C may be hardly visible to the naked eye. This is especially true for uninitialized structure fields, padding bytes and unions with diversely sized members. Nonetheless, there are also certain vulnerable code constructs specific to Windows – discussed in Sections 2.2.2 and 2.2.3 – which are easier to spot during code review. In May 2017, we decided to test this assumption by performing a cursory manual analysis of the Windows kernel in search of *low hanging fruit* issues that could have been missed by Bochspwn Reloaded due to incomplete code coverage.

On Linux, a good starting point is to locate references to the `copy_to_user` function and begin the audit from there, progressively going back through the code to track the initialization of each part of the copied objects. On Windows, there is no such dedicated function, and all user↔kernel memory operations are achieved with direct pointer manipulation or regular `memcpy` calls. However, usage of the `ProbeForWrite` function [89] may be a valuable hint, because pointer sanitization often directly preceeds writing data back to userland. To reduce the scope of the review, we decided to focus on the top-level syscall handlers, since they typically have a concise execution flow and easy to understand lifetime of objects in memory. Table 7 shows the combined numbers of direct and inlined `memcpy` calls (as detected by IDA Pro) in `Nt` functions on 32-bit builds of Windows 7 and 10, with the January 2018 patch installed. Assuming that approximately half of the calls were used to copy output data, we assessed the extent of required manual analysis to be manageable within several days.

	Windows 7	Windows 10
Core (`ntoskrnl.exe` etc.)	139	165
Graphics (`win32k.sys` etc.)	288	405

Table 7: Numbers of `memcpy` calls in Windows syscall handlers

While examining the system call implementations, we discovered two new vulnerabilities, now assigned CVE-2017-8680 and CVE-2017-8681. They were canonical examples of the "arbitrary output length" and "fixed-sized array" variants discussed earlier in this paper. The pseudo code of the affected `win32k.sys` routines is shown in Listings 23 and 24.

In the case of the first bug, the `cbBuffer` argument was used to request a pool allocation of that size in line 6, and the entire memory region was subsequently copied back to userland in line 12. The problem was that the dynamic memory was not zero-initialized upon allocation, and more importantly, that the syscall disregarded the actual number of bytes written to the buffer by `GreGetGlyphOutlineInternal`, instead only operating on the user-provided size. The leak had relatively high severity, as it allowed the attacker to disclose arbitrarily many bytes from allocations of controlled lengths. It also demonstrated that spotting a system call parameter passed directly to the 3rd argument of `memcpy` is frequently a red flag, as it indicates that a controlled number of bytes is returned to the caller instead of only the relevant output.

In the second bug, a temporary local array of 256 bytes (128 wide characters) was used to store a textual monitor description, later copied to the caller. The buffer was not pre-initialized, and was copied to user-mode as a whole regardless of the actual length of the string. In our test environment, this resulted in a disclosure of exactly 128 bytes from the kernel stack, but the number could be

```
DWORD NTAPI NtGdiGetGlyphOutline(
  ...,
  DWORD cbBuffer,
  LPVOID lpvBuffer
) {
  LPVOID KernelBuffer = Allocate(cbBuffer);

  DWORD Status = GreGetGlyphOutlineInternal(..., KernelBuffer, cbBuffer);

  if (Status != GDI_ERROR) {
    ProbeForWrite(lpvBuffer, cbBuffer, 1);
    memcpy(lpvBuffer, KernelBuffer, cbBuffer);
  }

  Free(KernelBuffer);

  return Status;
}
```

Listing 23: Vulnerable `win32k!NtGdiGetGlyphOutline` function pseudo code

```
NTSTATUS NTAPI NtGdiGetPhysicalMonitorDescription(
  HANDLE hMonitor,
  DWORD  dwSizeInChars,
  LPWSTR szDescription
) {
  WCHAR KernelBuffer[PHYSICAL_MONITOR_DESCRIPTION_SIZE];

  if (dwSizeInChars != PHYSICAL_MONITOR_DESCRIPTION_SIZE) {
    return STATUS_INVALID_PARAMETER;
  }

  NTSTATUS Status = CMonitorAPI::GetMonitorDescription(
      hMonitor, PHYSICAL_MONITOR_DESCRIPTION_SIZE, KernelBuffer);

  if (NT_SUCCESS(Status)) {
    ProbeForWrite(szDescription, sizeof(KernelBuffer), 1);
    memcpy(szDescription, KernelBuffer, sizeof(KernelBuffer));
  }

  return Status;
}
```

Listing 24: Vulnerable `win32k!NtGdiGetPhysicalMonitorDescription` function pseudo code

```
  ms_exc.registration.TryLevel = -2;
  if ( v16 )
    memset(v16, 0, a5);
  v10 = v16;
  v15 = GreGetGlyphOutlineInternal(a1, a2, a3, &v13, a5, v16, &v14, a8);
```

Listing 25: Diff of `win32k!NtGdiGetGlyphOutline` in Windows 7 and 10

different depending on the model of the physical monitor.

Both vulnerabilities were fixed by Microsoft in the September 2017 bulletin by adding adequate `memset` calls to erase the affected memory regions prior to operating on them. In conclusion, our experiment showed that manual code review may be an effective approach to finding kernel infoleaks which could be otherwise missed by other techniques.

5.3 Cross-version kernel binary diffing

At the time of this writing, there are three main versions of Windows under active support – Windows 7, 8.1 and 10 [80]. While Windows 7 and 8.1 combined still have just over 50% of the worldwide Windows market share [103], Microsoft is known for introducing a number of structural security improvements and sometimes even ordinary bugfixes only to the most recent Windows platform. This makes it potentially possible to detect 0-day vulnerabilities in the older systems merely by spotting subtle changes in the corresponding code in different versions of Windows.

When filing an issue for CVE-2017-8680 in the Project Zero bug tracker [60], we realized that the bug only affected Windows 7 and 8.1, while it had been internally fixed by Microsoft in Windows 10. Listing 25 shows the difference between the vulnerable and patched versions of the code, as decompiled by Hex-Rays and compared by Diaphora. The patch is evident, as a new `memset` call added in a top-level syscall handler is very likely a fix for a memory disclosure issue.

Based on this discovery, we suspected that there could be more such extra `memset` references in newer systems, revealing unpatched bugs in older ones. To verify this, we compared [52] decompiled listings of the `ntoskrnl.exe` and `win32k.sys` modules[10] between Windows 7, 8.1 and 10, looking for discrepancies in the usage of `memset`. The numbers of added invocations of the function in respective kernels are summarized in Table 8.

[10]Including `tm.sys`, `win32kbase.sys` and `win32kfull.sys` on Windows 10.

	`ntoskrnl.exe`		`win32k.sys`	
	functions	syscalls	functions	syscalls
Windows 7 vs. 10	153	8	89	16
Windows 8.1 vs. 10	127	5	67	11

Table 8: New `memset` calls in Windows 10 in relation to older systems

We then manually reviewed all instances of `memset` calls added to system call handlers. As a result, we discovered two new vulnerabilities:

- **CVE-2017-8684** – A stack-based disclosure of about 88 bytes in `win32k!NtGdiGetFontResourceInfoInternalW`, affecting Windows 7 and 8.1. It was a variant of the "arbitrary output length" pattern discussed in Section 2.2.3.
- **CVE-2017-8685** – A stack-based disclosure of 1024 bytes in `win32k!NtGdiEngCreatePalette`, affecting Windows 7.

The diffs that enabled us to recognize the vulnerabilities are shown in Listings 26 and 27.

```
v12 = 0;
v13 = 0;
memset(&v14, 0, 0x5Cu);
v11 = 0;
ms_exc.registration.TryLevel = 0;
```

Listing 26: Diff of `win32k!NtGdiGetFontResourceInfoInternalW` between Windows 7 and Windows 10

```
v16[0] = 0;
memset(&v16[1], 0, 0x3FCu);
v14 = 0;
if ( a2 > 0x10000 )
  return 0;
```

Listing 27: Diff of `win32k!NtGdiEngCreatePalette` between Windows 7 and Windows 8.1

In July 2017, we learned that another flaw could have been also discovered the same way – CVE-2017-11817, a leak of over 7 kB of uninitialized kernel pool memory to NTFS metadata, discussed in Section 6.1.2. The vulnerability was present in Windows 7, but a `memset` function had been added to `Ntfs!LfsRestartLogFile` in Windows 8.1 and later, potentially exposing the bug to researchers proficient in binary diffing. When reported to Microsoft, it was patched in the October 2017 Patch Tuesday.

Interestingly, we have also observed the opposite situation, where a `memset` call was removed from the kernel in newer systems, thus introducing a bug.

This was the case for CVE-2018-0832, a stack-based disclosure of four bytes in `nt!RtlpCopyLegacyContextX86` [64]. The buffer created with `alloca` was correctly zero-initialized in Windows 7, but not in Windows 8.1 or 10. The root cause behind adding the regression is unclear.

As demonstrated above, most fixes for kernel infoleaks are obvious both when seen in the source code and in assembly. The binary diffing required to identify inconsistent usage of `memset` doesn't require much low-level expertise or knowledge of operating system internals. Therefore, we hope that these bugs were some of the very few instances of such easily discoverable issues, and we encourage software vendors to make sure of it by applying security improvements consistently across all supported versions of their software.

5.4 Differential syscall fuzzing

In Section 2.5, we mentioned that several authors have proposed multi-variant program execution to identify use of uninitialized memory. The concept was implemented in various forms in the DieHard [21], BUDDY [44] and SafeInit [94] projects. We have found that a similar technique can be used to effectively identify Windows kernel infoleaks. If each newly allocated stack and pool-based region is filled with a single-byte pattern that changes over time, then a user-mode program may analyze the output of two subsequent syscall invocations looking for unequal, but repeated bytes at common offsets. Every occurrence of such a pattern most likely manifests an information disclosure vulnerability.

Filling stacks and pools with marker bytes may be accomplished in a few different ways. For pools:

1. The analyzed system may be run under the Bochs emulator, with an instrumentation that sets the bytes of all new allocations once they are requested.

2. As explained in Section 4 "Windows bug reproduction techniques", the desired effect is a part of the default behavior of the Special Pool option in Driver Verifier [79]. When the feature is enabled for any specific module, all pool regions returned to that module are filled with a marker byte (which changes after each new allocation).

3. Low-level hooks may be installed on kernel functions such as `ExAllocate-PoolWithTag` to briefly redirect code execution and set the allocation's bytes. This option was used by fanxiaocao and pjf [27]. It may be problematic on 64-bit Windows platforms due to the mandatory Patch Guard mechanism.

In turn, the following methods may be applied to the stack:

1. Again, the analyzed system may be run under Bochs, with an instrumentation responsible for writing markers to allocated stack frames (as described in Section 3.1.2 "Tainting stack frames").

2. Stack-spraying primitives (as detailed in Section 4 "Windows bug reproduction techniques") may be used prior to invoking the tested system calls, each time with a different value used for the spraying.

3. Low-level hooks may be installed on system call entry points such as `nt!KiFastCallEntry`, to poison the kernel stack before passing execution to the syscall handlers. This option was used by fanxiaocao and pjf [27], and similarly to their pool hooks, it may not work well with Patch Guard on 64-bit system builds.

Options (1) have the best allocation coverage, especially in terms of stack poisoning, but come at the cost of a significant slowdown. Options (2) have marginally worse coverage, but can be used on bare metal or in virtual machines, without digging in low-level system internals. Options (3) can be useful when a custom memory-poisoning mechanism is needed, and system performance plays a substantial role. We have successfully tested methods (1) and (2) and confirmed their effectiveness in discovering Windows security flaws.

Besides poisoning all newly requested allocations, it is necessary to develop a user-mode harness to invoke the tested syscalls and analyze their output. This is strongly related to the more general field of effective Windows system call fuzzing, which is still an open problem. Windows 10 RS3 32-bit supports as many as 460 native syscalls [66] and 1174 graphical ones [65]. Ideally, the harness should be aware of the prototypes of all system calls in order to invoke them correctly, reach their core functionality and interpret the output. To our best knowledge, there isn't currently any existing framework that would enable us to run this kind of analysis. As a result, we decided to take a more basic approach and focus on a specific subset of the system calls.

As we were looking for disclosure of uninitialized memory, we were primarily interested in services designed to query information and return it back to the caller. One such family of syscalls consists of kernel functions whose names start with the `NtQuery` prefix. Their purpose is to obtain various types of information regarding different objects and resources present in the system. There is currently a total of 60 such syscalls, with each type of kernel object having a corresponding service, e.g. `NtQueryInformationProcess` for processes, `NtQueryInformationToken` for security tokens, `NtQuerySection` for sections and so forth. Conveniently, a majority of these system calls share a common definition. An example prototype of the `NtQueryInformationProcess` handler is shown in Listing 28.

```
NTSTATUS WINAPI NtQueryInformationProcess(
  _In_      HANDLE           ProcessHandle,
  _In_      PROCESSINFOCLASS ProcessInformationClass,
  _Out_     PVOID            ProcessInformation,
  _In_      ULONG            ProcessInformationLength,
  _Out_opt_ PULONG           ReturnLength
);
```

Listing 28: Prototype of the `nt!NtQueryInformationProcess` system call

The parameters, in the original order, are:

- `ProcessHandle` – a handle to the queried object of a specific type, in this case a process.
- `ProcessInformationClass` – a numeric identifier of the requested information in the range of $[0..N]$, where N is the maximum supported information class on the given operating system. The names of some information classes can be found in Windows SDK header files, but few of them are officially documented. The classes are also subject to change between different versions of the operating system. In our experiment, we regarded the argument as a "black box" and brute-forced all possible values in the conservative range of [0..255]. Non-supported classes were filtered out based on the `STATUS_INVALID_INFO_CLASS` return code.
- `ProcessInformation` – a pointer to a user-mode buffer receiving the output data.
- `ProcessInformationLength` – length of the supplied output buffer. Every information class expects the length to be equal to the size of the output object(s), or to fall within a specific range. As the extent of correct values is limited, the argument can also be successfully brute-forced, or alternatively set according to the value returned through `ReturnLength`.
- `ReturnLength` – a pointer to a variable which receives the size of the requested information.

While the basic premise is shared across all `NtQuery` services, some of them may diverge slightly from the prototype shown above. For example, they may accept textual paths instead of object handles, use the `IO_STATUS_BLOCK` structure instead of a simple `ReturnLength` variable, or not take the information class as an argument, because only one type of data is returned.

We briefly reverse-engineered all 60 system calls in question, and determined that 31 of them were either simple enough that they didn't require dynamic testing, or could only be accessed by users with administrative rights. Accordingly, we developed test cases (in the form of standalone test programs) for the remaining 29 non-trivial syscalls. By running them on Windows 7 and 10 (32/64-bit) in both regular VMs and in Bochspwn Reloaded, we discovered infoleaks in a total of 14 services across 23 different information classes. The results of the experiment are summarized in Table 9. A majority of the bugs were stack-based leaks caused by uninitialized fields and padding bytes in the output structures.

We believe that the experiment demonstrates that differential syscall fuzzing may be effective in identifying kernel memory disclosure, and that certain groups of system calls are more suspectible to the problem than others due to their design and purpose.

System call (`NtQuery...`)	Information class
`AttributesFile`	—
`DirectoryFile`	`FileBothDirectoryInformation` (3) `FileIdBothDirectoryInformation` (37)
`FullAttributesFile`	—
`InformationJobObject`	`JobObjectBasicLimitInformation` (2) `JobObjectExtendedLimitInformation` (9) `JobObjectNotificationLimitInformation` (12) `JobObjectMemoryUsageInformation` (28)
`InformationProcess`	`ProcessVmCounters` (3) `ProcessImageFileName` (27) `ProcessEnergyValues` (76)
`InformationResourceManager`	`ResourceManagerBasicInformation` (0)
`InformationThread`	`ThreadBasicInformation` (0)
`InformationTransaction`	`TransactionPropertiesInformation` (1)
`InformationTransactionManager`	`TransactionManagerRecoveryInformation` (4)
`InformationWorkerFactory`	`WorkerFactoryBasicInformation` (7)
`Object`	`ObjectNameInformation` (1)
`SystemInformation`	`SystemPageFileInformation` (18) `MemoryTopologyInformation` (138) `SystemPageFileInformationEx` (144)
`VirtualMemory`	`MemoryBasicInformation` (0) `MemoryMappedFilenameInformation` (2) `MemoryImageInformation` (6) `MemoryPrivilegedBasicInformation` (8)
`VolumeInformationFile`	`FileFsVolumeInformation` (1)

Table 9: Vulnerable combinations of `NtQuery` syscalls and information classes

5.5 Taintless Bochspwn-style instrumentation

A full system instrumentation similar to Bochspwn Reloaded is capable of detecting some instances of disclosure of uninitialized memory even without the notion of shadow memory and taint tracking, as long as it is able to examine all user-mode memory writes originating from the kernel. One way to achieve this is to analyze all syscall output data in search of information that shouldn't normally be found there and thus manifests infoleaks – for example, valid ring 0 addresses. As mentioned earlier in the paper, kernel addresses are among the most common types of data found in uninitialized memory, and since they are also relatively easy to recognize (especially on 64-bit platforms), they may be used as highly reliable bug indicators. On the downside, the approach cannot detect leaks that don't contain any addresses, or leaks whose individual continuous chunks are smaller than the width of a pointer.

To improve the above design, it is possible to poison the stack and heap/pool allocations with a specific marker byte, and search for sequences of that byte in data written by the kernel to ring 3. This is trivial to achieve from the level of Bochs instrumentation, but might be more difficult when running the tested system in a regular virtual machine. The various available avenues of poisoning newly created kernel memory areas are detailed in Section 5.4 "Differential syscall fuzzing". An important quality of this technique is that it overcomes any potential limitations of taint tracking and propagation, as it is based solely on the analysis of actual memory contents in the guest system. On the other hand, it may be prone to false-positives, in cases where legitimate syscall output data happens to contain a sequence of the marker bytes. However, that risk can be significantly reduced by running several instances of the instrumentation with different marker bytes, and cross-checking the results to only analyze reports which reproduce across all of the sessions.

The taintless approach was successfully employed by fanxiaocao and pjf of IceSword Lab to discover 14 Windows kernel infoleaks in 2017 [27], and by the grsecurity team to identify an unspecified number of bugs in the Linux kernel in 2013 [33]. We also used a variant of this method to hunt for disclosure of uninitialized memory to mass-storage devices, as discussed in Section 6.1.

6 Other data sinks

In addition to the user-mode address space, uninitialized kernel memory may become available to unauthorized code through other data sinks, such as mass-storage devices (internal and external hard drives, USB flash drives, DVDs etc.) and the network. The UniSan tool [45] accounted for those possibilities in Linux by including the `sock_sendmsg` and `vfs_write` functions in the list of sinks together with `copy_to_user`, hence universally intercepting most exit points where data escapes the kernel. The idea proved effective, as more than 50% of new kernel vulnerabilities discovered by the project (10 out of 19) leaked memory through the socket sink.

In our experimentation, we focused on recognizing disclosure of Windows kernel memory through common file systems, using an enhanced variant of the taintless technique discussed in Section 5.5. In the following subsections, we outline the inner workings of this side project and present the results, including one particularly interesting vulnerability found in the `ntfs.sys` file system driver (CVE-2017-11817).

6.1 Mass-storage devices

Storage devices are a peculiar type of data sinks for kernel infoleaks, as any realistic attack which involves them requires physical access to the target machine and/or user interaction. Imagine, for example, that person A shares some files with person B on a flash drive. In the presence of kernel memory leaks to that drive, person A may believe that they are only sharing the contents of the few explicitly copied files, while in fact they may also be disclosing fragments of leftover data from their operating system. Furthermore, such unknowingly leaked memory saved to disks of various forms could be used as a source of forensic evidence, potentially revealing some of the actions performed by the user while having the storage devices connected to their computer. The severity and practical usefulness of such bugs is limited in comparison to leaks over the network or the kernel/user memory boundary, but we believe they still require proper attention from operating system vendors.

Some operating systems, such as Windows, automatically mount the partitions found on storage devices attached to the machine, even when the system is locked. Consequently, any memory that is leaked immediately upon mounting the volume opens up the possibility to automatically exfiltrate data through USB or other physical ports, bypassing the requirement of user interaction.

6.1.1 Detection

A canonical approach to sanitizing memory saved to disk in the Windows kernel would be to develop a file system filter driver, and have it intercept all *write* operations performed on the mounted volumes. Then, the driver would invoke a specially designed *hypercall*, which would send a signal to the instrumentation indicating that the taint of a specific memory area needs to be checked. However, instead of going that route, we decided to test a more experimental method of taintless detection, based on poisoning all new allocations in the system with a known marker byte, and scanning the guest's disk image in search of those markers. The technique was potentially more effective as it was not subject to the innacuracy of taint propagation, but it came at the cost of losing some valuable context information – we no longer learned about the call stack and system state at the exact time of the leak taking place. The only information available to us was the location of the disclosed bytes on disk, and the type of the leaked memory, if we used different markers for the stack and the pools.

This put us in an inconvenient spot where we could detect leaks, but it was difficult to analyze them and determine their root cause. In order to solve this problem, we tried to use the marker bytes to encode more information than just the fact that they originated from a kernel allocation – specifically, the addresses of the code that made the allocations. As the testing was performed on x86 versions of Windows, the pointers were 4-byte wide. Thus, a 16-byte pool buffer allocated at address `0x8b7ad4ab` would no longer be padded with the `0xaa` byte:

```
aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
```

but instead would be filled with a repeated address of its origin in little-endian format:

```
ab d4 7a 8b ab d4 7a 8b ab d4 7a 8b ab d4 7a 8b
```

Addresses of the allocation origins encountered by the instrumentation were saved to an output file, and later used by a separate script to periodically scan the disk image in search of kernel infoleaks. One evident shortcoming was that only leaks of a minimum of 4 bytes could be detected this way, but we considered this an acceptable trade-off for the additional information gained for the larger leaks.

The above method is probabilistic and heavily relies on the disk not containing the "magic" sequences of bytes in unrelated contexts. While the design is susceptible to false positives, we took steps to reduce their extent. Firstly, we changed the modes of the virtual disks subject to testing from `flat` to `volatile` in the `bochsrc` configuration file. This caused Bochs to save all disk modifications to a separate changelog file, thus enabling us to look for the markers in the disk delta instead of the entire multi-gigabyte image, which vastly decreased the chance of finding unrelated sequences of bytes colliding with our markers.

Secondly, we observed that values resembling valid kernel-mode addresses still occurred often in the changelog file, introducing noise in the output of our tool. Through trial and error, we determined that collisions happened much less frequently when the origins were first combined (xored) with another fixed 32-bit value, in our case `0x3bdd78ec` for pool markers and `0x7345c6f1` for stack markers. By applying this logic to the previous example, the allocation would be pre-initialized with bytes shown on the next page.

```
47 ac a7 b0 47 ac a7 b0 47 ac a7 b0 47 ac a7 b0
```

With the above scheme implemented, we were able to successfully identify uninitialized kernel memory written to persistent storage. Thanks to the encoded origin addresses, we could also investigate the related code and usually understand the root cause of the issues. While the available information was still limited, it allowed Microsoft to fix all of the reported bugs. Any remaining false positives were filtered out either by ignoring addresses of kernel functions completely unrelated to file system handling, or manually inspecting bytes surrounding the marker sequences in the changelog file to determine if they were legitimate leaks or accidental collisions.

6.1.2 Testing and results

In this experiment, we tested 32-bit builds of Windows 7 and 10, combined with file systems such as FAT, FAT32, exFAT and NTFS. We scanned both the system volume and an additional, separate partition created and attached to the guest system to determine which leaks only occur on the system drive, and which ones can be triggered by connecting an external device. Once the emulated OS was booted, we performed a series of operations on the file systems, including:

- Creating and deleting files of various sizes,
- Creating and deleting deeply nested directory structures,
- Renaming files and directories,
- Traversing the existing directory structure,
- Reading from and writing to existing files,
- Enabling and disabling compression and encryption (where applicable),
- Modifying the *hidden*, *read-only*, and security attributes of files and directories.

As a result of the testing, we didn't identify any issues in the handling of file systems from the FAT family. On the other hand, we discovered a multitude of both stack-based and pool-based leaks to NTFS volumes, originating from 10 different locations in the `ntfs.sys` driver (see Table 10). The size of the leaks was generally modest and ranged between 4-12 bytes in continuous blocks of uninitialized memory. Most of them affected all currently supported versions of Windows. They were initially submitted to Microsoft in a single report on July 7, 2017, followed by an update with more details on the bugs on August 25. A summary of our communication with MSRC can be found in the corresponding bug #1325 in the Project Zero bug tracker [58]. The issues were collectively fixed in a single bulletin as CVE-2017-11880 in November 2017.

Leaked allocation origin	Memory	Windows
`ntfs!NtfsInitializeReservedBuffer+0x20`	Pool	7
`ntfs!NtfsAddAttributeAllocation+0xb16`	Pool	7-10
`ntfs!NtfsCheckpointVolume+0xdcd`	Pool	7-10
`ntfs!NtfsDeleteAttributeAllocation+0x12d`	Pool	7-10
`ntfs!CreateAttributeList+0x1c`	Pool	7-10
`ntfs!NtfsCreateMdlAndBuffer+0x95`	Pool	7-10
`ntfs!NtfsDeleteAttributeAllocation+0xf`	Stack	7-10
`ntfs!NtfsWriteLog+0xf`	Stack	7-10
`ntfs!NtfsAddAttributeAllocation+0xf`	Stack	7-10
`ntfs!NtfsCreateAttributeWithAllocation+0xf`	Stack	7-10

Table 10: A summary of minor kernel infoleaks found in the `ntfs.sys` driver

One other, particularly dangerous vulnerability found by the project was a pool-based disclosure of about 7500 bytes in the `ntfs!LfsRestartLogFile` function [61], addressed by Microsoft separately in October 2017 as CVE-2017-11817. The severity of the flaw stemmed from the fact that it resulted in saving a very substantial amount of old kernel memory to the `$LogFile` pseudo-file, and it didn't require user interaction – the bug was triggered automatically every time a new NTFS volume was mounted in the system. Coupled with the fact that Windows mounts all connected storage devices even when the system is locked, this enabled attackers with physical access to the victim's powered-on machine to exfiltrate uninitialized kernel memory through the USB port. Remarkably, the issue only affected Windows 7, as it had been internally fixed in Windows 8 and later by adding a `memset` call to reset the affected memory area. The problems associated with inconsistent patching across supported system versions are explained in Section 5.3 "Cross-version kernel binary diffing".

6.2 Outbound network traffic

Network traffic is another example of a potential data sink for uninitialized kernel memory. It is arguably more important than mass-storage devices, as any such network-related bug could allow fully remote information disclosure, not otherwise possible through the other discussed channels. On Windows, the detection of such leaks could be implemented by developing a network filter driver to mark all packets passing through for sanitization by the instrumentation. Another option is a taintless approach similar to the one discussed in the previous section; poisoning all kernel allocations with a recognizable byte pattern, and searching for those patterns in the network traffic captured during the testing of the target system. We didn't perform any network-related experimentation during this project, and as such it remains an open research area.

7 Future work

Disclosure of uninitialized memory is far from a solved problem, and there are multiple avenues for improvement on all levels of work – both specific to Bochspwn Reloaded and the wider topic of mitigating this class of vulnerabilities in software. They are discussed in the paragraphs below.

Accurate taint tracking. In our implementation, taint was propagated only for memory copied with the `memcpy` function or its inlined equivalent (the `rep movs` instruction). This limitation may generate false negatives in cases where data is copied indirectly through CPU registers, which is becoming a common optimization scheme in modern compilers. This is especially relevant to Windows 10, where a considerable fraction of `memcpy` calls with constant sizes are compiled as a series of `mov` instructions between memory and registers. By extending the taint tracking logic to also cover registers without compromising on performance, the instrumentation could track memory and detect information leaks more effectively.

Extended code coverage. In dynamic binary instrumentation, the extent of code executed in the tested software marks the upper bound for the volume of issues that can be identified with this method. In the recent years, coverage-guided fuzzers have been trending in the security community, with the two canonical examples being american fuzzy lop [77] for user-mode applications and syzkaller [11] for the Linux kernel. Such fuzzers are able to progressively build a growing set of test cases reaching new code paths in the program, and thus greatly supplement any accompanying instrumentation. Likely due to the closed-source nature of Windows and related difficulty of implementation, there currently isn't a corresponding way of achieving vast kernel code coverage in that system. Any such project would instantly increase the bug detection rate of Bochspwn Reloaded and similar tools by a large margin.

Other data sinks. As discussed in Section 6, kernel memory may be disclosed not only through virtual memory, but also mass-storage devices and the network. While we briefly experimented with detecting infoleaks to file systems on Windows, the work is not complete, and there are still a number of untested system/data sink combinations and detection techniques left to be evaluated.

Other operating systems. During our experimentation, we tested 32-bit and 64-bit builds of Windows and a 32-bit build of Linux. The binary-level approach is most suitable for systems without publicly available source code, but as the Linux results show, it can also prove useful for platforms which in theory have more powerful debugging tools at their disposal. Therefore, we believe that Bochspwn Reloaded could be also succesfully used against other environments such as Linux x64, macOS and BSD-family systems.

Other security domains. We discussed how the characteristics of the C programming language contribute to the difficulty of securely passing data between different security domains in Section 2.1. In principle, the outlined problems are not specific to interactions between user space and the kernel, and may appear in any software where low-level objects are passed through shared memory between components with different privilege levels. In particular, we expect that both inter-process communication channels (used in sandboxing) and virtualization software may also be suspectible to similar infoleaks, and the concepts described in this paper should be applicable to these areas.

Other instrumentation types. To date, the two of our most productive full system instrumentation projects were designed to identify double fetch and memory disclosure bugs in OS kernels. However, we believe that the technique has much broader potential, and a number of other execution patterns can be formulated to detect security bugs or pinpoint sensitive areas of code. The ideas for various instrumentation types we recognized while working on Bochspwn Reloaded are documented in Appendix A. For some of them, such as "unprotected accesses to user-mode memory" or "double writes", we developed functional prototypes and used them to uncover several initial bugs, which shows that they are practical and may help expose further issues in the future.

Exploitation. Despite the amount of attention given to memory disclosure flaws, especially in the Linux community, little work has been done to investigate the exploitability of such bugs to achieve the disclosure of specific types of information, e.g. cryptographic keys, file contents, network traffic and so on. More research on the subject of manipulating the heap and pools to align the disclosed allocations with desired freed objects would help better understand the true severity of kernel infoleaks, beyond their utility in bypassing the KASLR and StackGuard mitigations.

Adoption of exploit mitigations. Extensive research has been performed on the subjects of detecting and mitigating infoleaks in the Linux kernel, as outlined in Section 2.5 "Prior research". We consider the most outstanding contributions to be UniSan [45], KMSAN [6] and grsecurity/PaX [4, 9]. Historically, most such projects were developed as independent tools and patches, and they were not included in the mainline kernel. We believe that Linux security would greatly benefit from integrating some of these features into the official code; more importantly, we also hope that the bug class will receive more recognition from Microsoft in the future. As the creator of Windows, the vendor is in a prime position to implement, evaluate and possibly deploy the techniques known in other operating systems. Due to the closed-source nature of Windows, many of these techniques are not otherwise available to external parties.

8 Conclusion

Information disclosure in general, and kernel memory disclosure in particular are a very specific class of software vulnerabilities, different from the more conventional types such as buffer overflows or use-after-free conditions. They don't reveal themselves through crashes or hangs, even though they may be triggered thousands of times every minute during normal system run time. As they are usually a by-product of functional user↔kernel communication and very difficult to spot in the code, they may remain unnoticed for many years.

In this paper, we elaborated on the different root causes and factors contributing to memory disclosure in modern software, and showed that the Windows operating system was affected by kernel infoleaks present in a range of drivers and system calls. To address the problem, we designed and implemented a Bochs-based instrumentation to automatically identify such issues during system run time. We then proceeded to use it to discover and report over 70 unique vulnerabilities in the Windows kernel and over 10 lesser bugs in Linux throughout 2017 and the beginning of 2018. To expand more on the general subject, we also evaluated alternative techniques for exposing similar leaks, and tackled the problem of recognizing uninitialized memory escaping the kernel to mass-storage devices. Finally, we reviewed other applications of system instrumentation to uncover kernel bugs and sensitive parts of the code.

We are optimistic that with continued work in the areas of detection and mitigation by OS vendors and compiler developers, the entire class of kernel infoleaks may be completely eliminated in the foreseeable future.

9 Acknowledgments

We would like to thank Gynvael Coldwind, Jann Horn, Joe Bialek, Matt Miller, Mathias Krause, Brad Spengler and Solar Designer for reviewing this paper or parts of it and providing valuable feedback.

References

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

[2] Building Modules. https://www.reactos.org/wiki/Building_Modules.

[3] Coccinelle. http://coccinelle.lip6.fr/.

[4] grsecurity. https://grsecurity.net/.

[5] KernelAddressSanitizer: Found Bugs. https://github.com/google/kasan/wiki/Found-Bugs.

[6] KernelMemorySanitizer. https://github.com/google/kmsan.

[7] KMSAN (KernelMemorySanitier) – Trophies. https://github.com/google/kmsan#trophies.

[8] Linux Test Project. https://linux-test-project.github.io/.

[9] PaX. https://pax.grsecurity.net/.

[10] Process Explorer. https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer.

[11] syzkaller - kernel fuzzer. https://github.com/google/syzkaller.

[12] The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[13] Trinity. http://codemonkey.org.uk/projects/trinity/.

[14] *ISO/IEC 9899:201x Committee Draft N1570*, April 2011.

[15] a_d_13. MemMAP v0.1.2. http://www.woodmann.com/collaborative/tools/index.php/MemMAP.

[16] Adrian Salido. dm ioctl: prevent stack leak in dm ioctl call. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4617f564c06117c7d1b611be49521a4430042287.

[17] Alexander Levin. kmemcheck: remove annotations. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4950276672fce5c241857540f8561c440663673d.

[18] Alexander Popov. How STACKLEAK improves Linux kernel security. https://linuxpiter.com/en/materials/2344. Linux Piter 2017.

[19] Alexander Potapenko. KernelMemorySanitizer against uninitialized memory. http://www.linuxplumbersconf.org/2017/ocw/proposals/4825.

[20] Alexandru Radocea, Georg Wicherski. Visualizing Page Tables for Local Exploitation: Hacking Like in the Movies. https://media.blackhat.com/us-13/US-13-Wicherski-Hacking-like-in-the-Movies-Visualizing-Page-Tables-Slides.pdf, https://media.blackhat.com/us-13/US-13-Wicherski-Hacking-like-in-the-Movies-Visualizing-Page-Tables-WP.pdf. Black Hat USA 2013.

[21] Berger, Emery D and Zorn, Benjamin G. DieHard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, volume 41, pages 158–168. ACM, 2006.

[22] Chen, Haogang and Mao, Yandong and Wang, Xi and Zhou, Dong and Zeldovich, Nickolai and Kaashoek, M Frans. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 5. ACM, 2011.

[23] Chow, Jim and Pfaff, Ben and Garfinkel, Tal and Rosenblum, Mendel. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *USENIX Security Symposium*, pages 22–22, 2005.

[24] Cowan, Crispan and Pu, Calton and Maier, Dave and Walpole, Jonathan and Bakke, Peat and Beattie, Steve and Grier, Aaron and Wagle, Perry and Zhang, Qian and Hinton, Heather. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[25] Dan Rosenberg. Subject: CVE request: multiple kernel stack memory disclosures. http://www.openwall.com/lists/oss-security/2010/09/25/2.

[26] Dan Rosenberg. Vulnerabilities. http://www.vulnfactory.org/vulns/. Listed under "memory disclosure".

[27] fanxiaocao and pjf of IceSword Lab (Qihoo 360). Automatically Discovering Windows Kernel Information Leak Vulnerabilities. http://www.iceswordlab.com/2017/06/14/Automatically-Discovering-Windows-Kernel-Information-Leak-Vulnerabilities_en/.

[28] fanxiaocao of IceSword Lab (Qihoo 360). great! I am also got multi case of "double-write". yet I report about 20 kernel pool address leak to MS. but they change the bar. https://twitter.com/TinySecEx/status/943410888119218176.

[29] fanxiaocao of IceSword Lab (Qihoo 360). new type of info-leak. https://twitter.com/TinySecEx/status/943417169953505282.

[30] fanxiaocao of IceSword Lab (Qihoo 360). you see , i am also found this case . haha! https://twitter.com/TinySecEx/status/943411731845410816.

[31] Feng Yan. *Windows Graphics Programming: Win32 GDI and DirectDraw.* Prentice Hall Professional, 2001.

[32] Feng Yuan. Source code for Windows Graphics Programming: Win32 GDI and DirectDraw. `https://blogs.msdn.microsoft.com/fyuan/2007/03/21/source-code-for-windows-graphics-programming-win32-gdi-and-directdraw/`.

[33] grsecurity. Probably didn't find anything in the typical Linux userland interface because 2013 we did some similar instrumentation to make some leaks fall out - modified the magic for STACKLEAK/SANITIZE to a value we told a fuzzer to never provide to the kernel,inspected copy_to_user for it. `https://twitter.com/grsecurity/status/991450745642905602`.

[34] Jann Horn. MacOS getrusage stack leak through struct padding. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1405`.

[35] Jon Oberheide. Advisories. `https://jon.oberheide.org/advisories/`. Listed under "Stack Disclosure".

[36] Jon Oberheide, Dan Rosenberg. Stackjackin' 2: Electric Boogaloo. `https://jon.oberheide.org/blog/2011/07/06/stackjackin-2-electric-boogaloo/`.

[37] Jon Oberheide, Dan Rosenberg. Stackjacking Your Way to grsec/PaX Bypass. `https://jon.oberheide.org/blog/2011/04/20/stackjacking-your-way-to-grsec-pax-bypass/`.

[38] Jonathan Corbet. Preventing kernel-stack leaks. `https://lwn.net/SubscriberLink/748642/4cdb84b99ce171e6/`.

[39] Joseph Bialek. Anyone notice my change to the Windows IO Manager to generically kill a class of info disclosure? BufferedIO output buffer is always zero'd. `https://twitter.com/JosephBialek/status/875427627242209280`.

[40] Juan Vazquez. Revisiting an Info Leak. `https://blog.rapid7.com/2015/08/14/revisiting-an-info-leak/`.

[41] Kees Cook. fork: unconditionally clear stack on fork. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e01e80634ecdde1dd113ac43b3adad21b47f3957`.

[42] Ken Johnson, Matt Miller. Exploit Mitigation Improvements in Windows 8. `https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf`. Black Hat USA 2012, Slide 35.

[43] Kurmus, Anil and Zippel, Robby. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1366–1377. ACM, 2014.

[44] Lu, Kangjie. *Securing software systems by preventing information leaks.* PhD thesis, Georgia Institute of Technology, 2017.

[45] Lu, Kangjie and Song, Chengyu and Kim, Taesoo and Lee, Wenke. UniSan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 920–932. ACM, 2016.

[46] Lu, Kangjie and Walter, Marie-Therese and Pfaff, David and Nürnberger, Stefan and Lee, Wenke and Backes, Michael. Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying. In *NDSS'17, Network and Distributed System Security Symposium*, 2017.

[47] Mateusz Jurczyk. A story of win32k!cCapString, or unicode strings gone bad. `http://j00ru.vexillium.org/?p=1609`.

[48] Mateusz Jurczyk. FreeType 2.5.3 CFF CharString parsing heap-based buffer overflow in "cff_builder_add_point". `https://bugs.chromium.org/p/project-zero/issues/detail?id=185`.

[49] Mateusz Jurczyk. FreeType 2.5.3 multiple unchecked function calls returning FT_Error. `https://bugs.chromium.org/p/project-zero/issues/detail?id=197`.

[50] Mateusz Jurczyk. nt!NtMapUserPhysicalPages and Kernel Stack-Spraying Techniques. `http://j00ru.vexillium.org/?p=769`.

[51] Mateusz Jurczyk. Subtle information disclosure in WIN32K.SYS syscall return values. `http://j00ru.vexillium.org/?p=762`.

[52] Mateusz Jurczyk. Using Binary Diffing to Discover Windows Kernel Memory Disclosure Bugs. `https://googleprojectzero.blogspot.com/2017/10/using-binary-diffing-to-discover.html`.

[53] Mateusz Jurczyk. Windows Kernel 64-bit stack memory disclosure in nt!NtQueryVirtualMemory (MemoryImageInformation). `https://bugs.chromium.org/p/project-zero/issues/detail?id=1519`.

[54] Mateusz Jurczyk. Windows Kernel Local Denial-of-Service #1: win32k!NtUserThunkedMenuItemInfo (Windows 7-10). `http://j00ru.vexillium.org/?p=3101`.

[55] Mateusz Jurczyk. Windows Kernel Local Denial-of-Service #2: win32k!NtDCompositionBeginFrame (Windows 8-10). `http://j00ru.vexillium.org/?p=3151`.

[56] Mateusz Jurczyk. Windows Kernel Local Denial-of-Service #3: nt!NtDuplicateToken (Windows 7-8). http://j00ru.vexillium.org/?p=3187.

[57] Mateusz Jurczyk. Windows Kernel Local Denial-of-Service #4: nt!NtAccessCheck and family (Windows 8-10). http://j00ru.vexillium.org/?p=3225.

[58] Mateusz Jurczyk. Windows Kernel multiple stack and pool memory disclosures into NTFS file system metadata. https://bugs.chromium.org/p/project-zero/issues/detail?id=1325.

[59] Mateusz Jurczyk. Windows Kernel pool memory disclosure due to output structure alignment in win32k!NtGdiGetOutlineTextMetricsInternalW. https://bugs.chromium.org/p/project-zero/issues/detail?id=1144.

[60] Mateusz Jurczyk. Windows Kernel pool memory disclosure in win32k!NtGdiGetGlyphOutline. https://bugs.chromium.org/p/project-zero/issues/detail?id=1267.

[61] Mateusz Jurczyk. Windows Kernel pool memory disclosure into NTFS metadata ($LogFile) in Ntfs!LfsRestartLogFile. https://bugs.chromium.org/p/project-zero/issues/detail?id=1352.

[62] Mateusz Jurczyk. Windows Kernel Reference Count Vulnerabilities - Case Study. http://j00ru.vexillium.org/slides/2012/zeronights.pdf. ZeroNights 2012.

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

[64] Mateusz Jurczyk. Windows Kernel stack memory disclosure in nt!RtlpCopyLegacyContextX86. https://bugs.chromium.org/p/project-zero/issues/detail?id=1425.

[65] Mateusz Jurczyk. Windows WIN32K.SYS System Call Table (NT/2000/XP/2003/Vista/2008/7/8/10). http://j00ru.vexillium.org/syscalls/win32k/32/.

[66] Mateusz Jurczyk. Windows X86 System Call Table (NT/2000/XP/2003/Vista/2008/7/8/10). http://j00ru.vexillium.org/syscalls/nt/32/.

[67] Mateusz Jurczyk. x86 Kernel Memory Space Visualization (KernelMAP v0.0.1). http://j00ru.vexillium.org/?p=269.

[68] Mateusz Jurczyk, Gynvael Coldwind. Bochspwn: Exploiting Kernel Race Conditions Found via Memory Access Patterns. http://j00ru.vexillium.org/slides/2013/syscan.pdf. SyScan 2013.

[69] Mateusz Jurczyk, Gynvael Coldwind. Bochspwn: Identifying 0-days via system-wide memory access pattern analysis. http://j00ru.vexillium.org/slides/2013/bhusa.pdf. Black Hat USA 2013.

[70] Mateusz Jurczyk, Gynvael Coldwind. Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns. http://vexillium.org/dl.php?bochspwn.pdf.

[71] Mateusz Jurczyk, Gynvael Coldwind. kfetch-toolkit. https://github.com/j00ru/kfetch-toolkit.

[72] Mateusz Jurczyk, Gynvael Coldwind. read_lin_mem() function. https://github.com/j00ru/kfetch-toolkit/blob/master/instrumentation/mem_interface.cc.

[73] Mathias Krause. CVE Requests (maybe): Linux kernel: various info leaks, some NULL ptr derefs. http://www.openwall.com/lists/oss-security/2013/03/05/13.

[74] Matt Tait. Google Project Zero Bug Tracker. https://bugs.chromium.org/p/project-zero/issues/list?can=1&q=id%3A390%2C435%2C453.

[75] Matt Tait. Kernel-mode ASLR leak via uninitialized memory returned to usermode by NtGdiGetTextMetrics. https://bugs.chromium.org/p/project-zero/issues/detail?id=480.

[76] Michal Hocko. kmemcheck: rip it out for real. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f335195adf043168ee69d78ea72ac3e30f0c57ce.

[77] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[78] Microsoft. Acknowledgments 2015. https://docs.microsoft.com/en-us/security-updates/Acknowledgments/2015/acknowledgments2015.

[79] Microsoft. Special Pool (MSDN). https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/special-pool.

[80] Microsoft. Windows lifecycle fact sheet. https://support.microsoft.com/en-us/help/13853/windows-lifecycle-fact-sheet.

[81] Microsoft (MSDN). ExAllocatePoolWithQuotaTag function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-exallocatepoolwithquotatag.

[82] Microsoft (MSDN). ExAllocatePoolWithTag function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-exallocatepoolwithtag.

[83] Microsoft (MSDN). /GS (Buffer Security Check). https://msdn.microsoft.com/en-us/library/8dbf701c.aspx.

[84] Microsoft (MSDN). Handling Exceptions. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/handling-exceptions.

[85] Microsoft (MSDN). Microsoft public symbol server. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/microsoft-public-symbols.

[86] Microsoft (MSDN). MmSecureVirtualMemory function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-mmsecurevirtualmemory.

[87] Microsoft (MSDN). ObReferenceObjectByHandle function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-obreferenceobjectbyhandle.

[88] Microsoft (MSDN). ProbeForRead function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-probeforread.

[89] Microsoft (MSDN). ProbeForWrite function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-probeforwrite.

[90] Microsoft (MSDN). StackWalk64 function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms680650(v=vs.85).aspx.

[91] Microsoft (MSDN). SymFromAddr function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms681323(v=vs.85).aspx.

[92] Microsoft (MSDN). SymLoadModule64 function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms681352(v=vs.85).aspx.

[93] Microsoft (MSDN). Windows Kernel-Mode Object Manager. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-kernel-mode-object-manager.

[94] Milburn, Alyssa and Bos, Herbert and Giuffrida, Cristiano. SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)(San Diego, CA)*, 2017.

[95] MITRE. CWE-252: Unchecked Return Value. https://cwe.mitre.org/data/definitions/252.html.

[96] North, John. Identifying Memory Address Disclosures. 2015.

[97] Peiró, Salva and Muñoz, M and Masmano, Miguel and Crespo, Alfons. Detecting stack based kernel information leaks. In *International Joint Conference SOCO14-CISIS14-ICEUTE14*, pages 321–331. Springer, 2014.

[98] Peiró, Salva and Munoz, M and Crespo, Alfons. An analysis on the impact and detection of kernel stack infoleaks. *Logic Journal of the IGPL*, 24(6):899–915, 2016.

[99] pj4533. MemSpyy. https://www.codeproject.com/Articles/21090/MemSpyy.

[100] Robert C. Seacord. *The CERT C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure Systems*. Addison-Wesley Professional, 2014. DCL39-C. Avoid information leakage when passing a structure across a trust boundary.

[101] Sebastian Apelt. Pwn2Own 2014 - Escaping the sandbox through AFD.sys. http://siberas.blogspot.de/2014/07/pwn2own-2014-escaping-sandbox-through.html.

[102] Solar Designer. Finally went through the Bochspwn Reloaded slides. Kudos! Feedback: rather than “Remove taint on free”, you could re-taint & detect UAF+leak. https://twitter.com/solardiz/status/879288169174315008.

[103] StatCounter GlobalStats. Desktop Windows Version Market Share Worldwide. http://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide.

[104] Tavis Ormandy. Fun Rev. Challenge: On 32bit Windows7, explain where the upper 16bits of eax come from after a call to NtUserRegisterClassEx-WOW(). https://twitter.com/taviso/status/16853682570.

[105] Tavis Ormandy. iknowthis Linux System Call Fuzzer. https://github.com/rgbkrk/iknowthis.

[106] Tess Ferrandez. Show me the memory: Tool for visualizing virtual memory usage and GC heap usage. https://blogs.msdn.microsoft.com/tess/2009/04/23/show-me-the-memory-tool-for-visualizing-virtual-memory-usage-and-gc-heap-usage/.

[107] Vasiliy Kulikov. Re: Linux kernel proactive security hardening. http://seclists.org/oss-sec/2010/q4/129.

[108] Vegard Nossum. Getting started with kmemcheck. https://www.kernel.org/doc/html/v4.12/dev-tools/kmemcheck.html.

[109] Wandering Glitch. Leaking Windows Kernel Pointers. https://ruxcon.org.au/assets/2016/slides/RuxCon%20-%20Leaking%20Windows%20Kernel%20Pointers.pdf.

[110] Weimin Wu. An Analysis of A Windows Kernel-Mode Vulnerability (CVE-2014-4113). https://blog.trendmicro.com/trendlabs-security-intelligence/an-analysis-of-a-windows-kernel-mode-vulnerability-cve-2014-4113/.

A Other system instrumentation schemes

During the development of Bochspwn and Bochspwn Reloaded, we have considered a number of alternative ways in which full system instrumentation could help identify security flaws, or at least signal sensitive areas of code that should receive more attention. For some of the ideas, we implemented functional prototypes which did uncover new bugs in the Windows kernel. While none of these experimental tools had as much success as the two main projects, we decided to discuss them in this appendix for completeness. We hope that the concepts outlined in the following subsections may serve as a source of inspiration for researchers aiming to take up the subject of kernel instrumentation through software emulation. They are mostly discussed in the context of Windows, but many of them also apply to Linux and other operating systems.

A.1 Enumeration of kernel attack surface

All interactions between user-mode and the kernel may be considered attack surface in itself – due to the numerous pitfalls described earlier in this paper – or more generally, reliable indicators of attack surface related to the processing of input data provided by the caller. Thanks to pointer annotations and dedicated copy functions, all such interactions in the Linux kernel may be enumerated using static code analysis.

The same objective cannot be achieved as easily on Windows, due to the lack of annotations and specialized functions, and the unavailability of the kernel source code. Without performing a thorough analysis of the assembly surrounding a specific memory access and all code paths leading up to it, it is often impossible to determine if an instruction operates on a user-mode pointer, a kernel-mode pointer, or both. There are reliable symptoms revealing the usage of ring 3 memory, such as references to the `ProbeForRead` and `ProbeForWrite` functions or the `MmUserProbeAddress` constant, but they are often inaccurate, as pointer sanitization may take place in a completely different location than the actual reference to that pointer.

To effectively enumerate the Windows kernel attack surface, a simple dynamic instrumentation may be used to detect and log all accesses to the user address space originating from the kernel. As with every instrumentation, its effectiveness is limited by the kernel code coverage reached during testing, but even just booting up the system is sufficient to generate enough logs for weeks of Windows kernel auditing.

A.2 Deeply nested user-mode memory accesses

In a majority of cases, reads and writes from/to user-mode memory should occur in top-level syscall handlers – `Nt` functions in Windows and `sys_` functions in Linux. As the first point of contact between ring $3 \leftrightarrow 0$, the routines should create local copies of input data, allocate temporary kernel buffers for output data, and take full responsibility for the interactions with userland, instead of burdening internal kernel functions with operating on user-mode pointers. With the lack of explicit pointer annotations in Windows, no kernel routine may definitely *know* if a pointer it receives as an argument is a user-mode one or not.

Passing along a user-controlled pointer into nested functions may blur the understanding of which part of the code is accountable for sanitizating the address. This may have dire consequences for system security, as illustrated in Listing 29.

```
NTSTATUS Bar(LPDWORD Output) {
  Output[0] = 0xdeadbeef;
  Output[1] = 0xbadc0ffe;
  Output[2] = 0xcafed00d;
  return STATUS_SUCCESS;
}

NTSTATUS Foo(DWORD Type, LPDWORD Output) {
  if (Type == 2) {
    return Bar(Output);
  }

  [...]
}

NTSTATUS NTAPI NtMagicValues(DWORD Family, DWORD Type, LPDWORD Output) {
  if (Family == 1) {
    return Foo(Type, Output);
  }

  [...]
}
```

Listing 29: Example of passing an input pointer to nested kernel functions

In this example, the `Output` parameter is never validated before being written to. The top-level `NtMagicValues` handler doesn't sanitize the pointer because it doesn't directly operate on it. The `Foo()` function doesn't do it, because it assumes that any argument it receives will already have been checked by the caller. Finally, the `Bar()` function doesn't do it, because it is a simple internal function that has no notion of different types of pointers. As a whole, this results in an arbitrary kernel memory overwrite – an easily exploitable security flaw – all because of the ambiguity caused by passing user-mode pointers to internal kernel functions which do not expect it.

Potential issues of this kind may be flagged by logging all user-mode memory references taking place within relatively deep callstacks. While not all instances of such behavior manifest actual bugs, heavily nested accesses to the ring 3 address space may suggest that the relevant code is not aware of the nature of the referenced pointer. This in turn increases the likelihood of the presence of a vulnerability, and warrants manual follow-up analysis.

A.3 Unprotected accesses to user-mode memory

Kernels should typically never assume the validity of user-mode pointers, unless the address range in question is explicitly locked in memory. In Windows, this imposes the need to wrap each user-mode memory reference with an adequate exception handler. The absence of such handler at any place where a controlled pointer is accessed may be exploited to trigger an unhandled exception and crash the operating system. The need to safely handle all exceptions arising from the usage of ring 3 memory is reflected in the documentation of the `ProbeForRead` function [88]:

> Drivers must call ProbeForRead inside a try/except block. If the routine raises an exception, the driver should complete the IRP with the appropriate error. Note that subsequent accesses by the driver to the user-mode buffer must also be encapsulated within a try/except block: a malicious application could have another thread deleting, substituting, or changing the protection of user address ranges at any time (even after or during a call to ProbeForRead or ProbeForWrite).

Moreover, the "Handling Exceptions" MSDN article [84] includes an adequate code example shown in Listing 30.

```
try {
    ...
    ProbeForWrite(Buffer, BufferSize, BufferAlignment);

    /* Note that any access (not just the probe, which must come first,
     * by the way) to Buffer must also be within a try-except.
     */
    ...
} except (EXCEPTION_EXECUTE_HANDLER) {
    /* Error handling code */
    ...
}
```

Listing 30: Example of a secure implementation accessing user-mode memory in the Windows kernel

From a technical standpoint, it is relatively easy to detect user-mode accesses being performed when no exception handlers are set up. Since the exception handling mechanism is largely different in 32-bit and 64-bit versions of Windows, we will focus on the subjectively easier goal of instrumenting the x86 platform. However, please note that each unsafe memory access identified on a 32-bit version of the system should reproduce on a 64-bit build, and vice-versa.

In x86 Windows builds, the handler records are chained together in a SEH (*Structured Exception Handling*) chain starting at the `fs:[0]` address, where each handler is described by the `_EH3_EXCEPTION_REGISTRATION` structure shown in Listing 31.

```
struct _EH3_EXCEPTION_REGISTRATION {
  struct _EH3_EXCEPTION_REGISTRATION *Next;
  PVOID ExceptionHandler;
  PSCOPETABLE_ENTRY ScopeTable;
  DWORD TryLevel;
};
```

Listing 31: Definition of the _EH3_EXCEPTION_REGISTRATION structure

The structures reside in the stack frames of their corresponding functions and are initialized by calling the __SEH_prolog4(_GS) procedures. During execution, entering the try{} blocks is denoted by writing their zero-based indexes to the TryLevel fields in the aforementioned structures, and later overwriting them with -2 (0xfffffffe) when execution leaves the blocks and exception handling is disabled. Below is an example of a try/except block encapsulating the writing of a single DWORD value to user space:

```
PAGE:00671CF3      mov     [ebp+ms_exc.registration.TryLevel], 1
PAGE:00671CFA      mov     eax, [ebp+var_2C]
PAGE:00671CFD      mov     ecx, [ebp+arg_14]
PAGE:00671D00      mov     [ecx], eax
PAGE:00671D02      mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFEh
```

It is possible for the Bochs instrumentation to iterate through the SEH chain by following the Next pointers, and determine which handlers are enabled and which functions they correspond to. If there are no exception records present, or all of them have their TryLevel fields set to -2, then an exception occurring right at that moment could potentially crash the operating system. It should be noted, however, that not all non-guarded accesses to user-mode memory are dangerous by definition – regions previously secured with MmSecureVirtualMemory [86] and special areas such as TEB or PEB are not affected.

In February 2017, we implemented the detection scheme and ran it against Windows 7 and 10. After analyzing the output, we identified several bugs allowing local attackers to trigger unhandled exceptions in the kernel and crash the operating system with a Blue Screen of Death. As denial-of-service issues don't meet the bar to be fixed by Microsoft in a security bulletin, we posted the details and proof-of-concept code of our findings on our blog [54, 55, 56, 57].

Affected syscall	Windows 7	Windows 8.1	Windows 10
win32k!NtUserThunkedMenuItemInfo	✓	✓	✓
win32k!NtDCompositionBeginFrame		✓	✓
nt!NtDuplicateToken	✓	✓	
nt!NtAccessCheck etc.	✓	✓	

Table 11: Summary of DoS bugs caused by unprotected access to user space

A.4 Broad exception handlers

In Windows, for every `try{}` block of code modifying any global data structures in the kernel, there should be a corresponding `except{}` block which reverts all persistent changes made prior to the exception. This is relatively easy to achieve with a flat structure of the code, when all relevant operations are explicitly visible inside the `try{}`. On the other hand, the rule is more difficult to enforce when nested calls are used, thus obfuscating the code flow and potentially facilitating the interruption of functions which don't anticipate being preempted in the middle of execution. In certain situations, this can lead to leaving global objects in the system in an inconsistent state, which may open up security vulnerabilities. One example of such bug is CVE-2014-1767 [101], a dangling pointer flaw in the `afd.sys` network driver, which was used during the pwn2own competition to elevate privileges in the system as part of a longer exploit chain.

```
typedef struct _DATA_ITEM {
  LIST_ENTRY ListEntry;
  DWORD Value;
} DATA_ITEM, *PDATA_ITEM;

PDATA_ITEM DataListHead;

VOID KeAddValueInternal(PDWORD ValuePtr) {
  PDATA_ITEM NewDataItem = ExAllocatePool(PagedPool, sizeof(DATA_ITEM));
  PDATA_ITEM OldListHead = DataListHead;

  DataListHead = NewDataItem;

  NewDataItem->Value = *ValuePtr;
  NewDataItem->ListEntry.Flink = OldListHead;
  NewDataItem->ListEntry.Blink = OldListHead->ListEntry.Blink;

  OldListHead->ListEntry.Blink = NewDataItem;
}

NTSTATUS NTAPI NtAddValue(PDWORD UserValuePtr) {
  NTSTATUS Status = STATUS_SUCCESS;

  try {
    ProbeForRead(UserValuePtr, sizeof(DWORD), sizeof(DWORD));
    KeAddValueInternal(UserValuePtr);
  } except (EXCEPTION_EXECUTE_HANDLER) {
    Status = GetExceptionCode();
  }

  return Status;
}
```

Listing 32: Example of a vulnerability caused by a broad exception handler

Let's consider an example shown in Listing 32. The `NtAddValue` top-level syscall handler validates the `UserValuePtr` pointer but doesn't read its value, instead passing it to the internal `KeAddValueInternal` function. The latter routine is not aware of the type of the pointer it receives, so it simply implements its self-contained logic – allocates a new object in memory, inserts it into a doubly-linked list and initializes it with the input data. A problem arises when accessing `ValuePtr` in line 14 fails and generates an `ACCESS_VIOLATION` exception, thus effectively aborting the execution of `KeAddValueInternal` and jumping straight into the handler in line 28. Due to the fact that the nested function was interrupted, it has already saved the newly allocated object as the head of the list, but hasn't yet initialized the object's `LIST_ENTRY` structure. On the other hand, `NtAddValue` doesn't know the internal logic of the functions it invokes, so it doesn't revert any changes made by them. Consequently, the linked list becomes corrupted with uninitialized pointers, leaving the system unstable and prone to privilege escalation attacks.

Candidates for such issues can be automatically identified by system instrumentation by examining all kernel accesses to user-mode memory[11] and checking the location of the first enabled exception handler in the stack trace for each of them. If the closest handler is not in the current function (or even worse, several functions below in the call stack), it is an indicator of a broad exception handler that could be unprepared to correctly restore the kernel to the state before the exception. That said, not all broad handlers are bugs, so each case needs to be examined on its own to determine if any changes made up to the current point of execution are not accounted for in the corresponding exception handler.

A.5 Dereferences of unsanitized user-mode pointers

To maintain system security, every user-mode pointer passed to the Windows kernel by a client application needs to be sanitized with a `ProbeForRead` [88] or `ProbeForWrite` [89] function (or equivalent) before being operated on. Missing pointer checks may create memory disclosure/corruption primitives such as arbitrary read/write, making them one of the most severe types of kernel vulnerabilities. If system instrumentation could recognize pointer sanitization at the OS runtime, it could then audit each access to user space to determine if the referenced address has been already sanitized in the context of the current system call, and report the bug if not.

One way to catch instances of pointer sanitization is to detect the execution of the `ProbeFor` functions and record the address ranges found in their arguments. The problem with this approach is that there are very few references to these functions in the core kernel images, especially to `ProbeForRead`. Instead of issuing direct calls to the function, its usages are typically inlined to the assembly equivalent of the pseudo-code shown on the next page.

[11]Or any other events that are known to generate exceptions, such as calls to `ExAllocatePoolWithQuotaTag` [81].

```
if ((Address & (~Alignment)) != 0) {
  ExRaiseDatatypeMisalignment();
}

if ((Address + Length < Address) ||
    (Address + Length >= MmUserProbeAddress)) {
  // Trigger an ACCESS_VIOLATION exception.
}
```

In some cases, the comparison with `MmUserProbeAddress` is performed against the value in memory, and in others it is achieved through a register. Small, constant values of `Length` are often ignored. Furthermore, the `win32k.sys` driver on Windows 7 and 8.1 doesn't use the `MmUserProbeAddress` constant exported from `ntoskrnl.exe`, but instead has its own copy stored in a static `W32UserProbeAddress` variable initialized in `DriverEntry`. To make things even more complicated, the constant is no longer used in the core kernel image of Windows 10 64-bit, as it was replaced with an immediate `0x7ffffff0000` operand in all of the relevant `cmp` instructions.

To address the above problems and reliably detect all instances of user-mode pointer sanitization, we propose the following logic:

- Intercept all `cmp` instructions with a register or memory as the first operand.
- Resolve the values of both operands of the instruction.
- If the value of the second operand is equal to `MmUserProbeAddress` (for example `0x7fff0000` in x86 builds), mark the address in the first operand as sanitized in the scope of the current syscall.

According to our experimentation, following the above steps in the Bochs instrumentation is sufficient to recognize pointer checking. Unfortunately, this still only provides us with very limited information in the form of just one end of the *probed* user-mode memory region (beginning or end, depending on the code construct). One solution to the problem of incomplete data is to extend the area marked as sanitized to the entire memory page the address belongs to, or even several pages around it. While this approach may significantly reduce the volume of false-positive results, it shifts the balance towards more false-negatives. For example, in this scheme, the sanitization of a single stack-based pointer would automatically cause all other pointers residing within the same page to also be considered as verified. This limitation can be partially mitigated by enabling the PageHeap mechanism for all user-mode programs run in the guest system, which should at least prevent heap-based pointers from overlapping with each other on the memory page level.

For testing purposes, we developed a basic prototype of the above scheme and tested it against Windows 7 and 10, but only identified references to unsanitized addresses in code accessible by system administrators or very early during system boot (e.g. in the `win32k!WmsgpConnect` function reachable through win32k!RegisterLogonProcess). However, there is much room for improvement in this field, and we believe that the instrumentation could successfully detect security issues provided an effective way to determine both ends of each probed ring 3 address range.

A.6 Read-after-write conditions

When the kernel writes a value to user-mode memory, there is no guarantee that it won't be modified the very next moment by a concurrent user thread. Thus, reading from a ring 3 memory region that was previously written to in the same system call is a strong indicator of a serious problem in the code. For example, such behavior could be manifested by a function assuming it is operating on a safe, kernel pointer, while in practice it was passed an untrusted, user-mode address. Such a scenario is illustrated in Listing 33.

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

Listing 33: Example read-after-write leading to a *write-what-where* condition

In lines 6-8, the syscall handler initializes the output UNICODE_STRING structure as an empty string backed by a user-mode buffer. Further on, it calls the RtlAppendUnicodeString API on that structure to fill it with a textual representation of the system version. The problem in the code is that the latter routine assumes that it receives a non-volatile kernel UNICODE_STRING object, while in fact it is passed a user-mode pointer whose data may change asynchronously at any point of the system call execution. A malicious program could use a concurrent thread to exploit the race condition by changing the value of UnicodeString->Buffer, to point it into the kernel address space within the short time window between the initialization of the pointer and its usage in the API function.

By nature, the bug class is very similar to double fetches, with the main difference being that the affected code trusts the contents of user-mode memory not because it has already read it once, but because it has explicitly initialized it to a specific value. The detection of such issues is also almost identical to the logic implemented in the original Bochspwn project [70], with the addition of instrumenting not only kernel→user memory reads, but also writes. We expect read-after-write conditions to be mostly specific to Windows, as it seems to be strongly tied to direct user-mode pointer manipulation and the lack of clear distinction between ring 3 and ring 0 pointers.

A.7 Double writes

System calls are meant to be invoked synchronously; only after execution returns to the user-mode client, should it read the output data provided by the kernel. However, if at any point the kernel writes a piece of privileged information to ring 3 and later overwrites it with legitimate data in the scope of the same syscall, a concurrent thread is able to obtain the initially written value within the time window of the race condition.

In theory, the bug class could apply to any type of sensitive data, but in practice it most often facilitates the disclosure of kernel-mode pointers, allowing attackers to bypass KASLR. The typical scenario for introducing a flaw of this kind is when a common structure containing pointers is used to store data both in kernel and user-mode. When a system call handler wishes to copy such an internal kernel structure to a client application, it is usually achieved by (1) copying the overall object in memory and (2) adjusting the particular pointers to point into the corresponding userland buffers. This may seem valid at first glance, but in fact it means that the original kernel-mode pointers reside in ring 3 for a brief period of time before they are overwritten with the appropriate addresses. As more effort is continuously being put in by OS vendors to protect information about the kernel address space, leaking ring 0 pointers may prove useful for the exploitation of other kernel memory corruption vulnerabilities.

```
typedef struct _USERNAME {
  UNICODE_STRING String;
  WCHAR Buffer[128];
} USERNAME, *PUSERNAME;

NTSTATUS NTAPI NtGetAdminUsername(PUSERNAME OutputUsername) {
  USERNAME LocalUsername;

  RtlZeroMemory(&LocalUsername, sizeof(USERNAME));

  StringCchCopy(LocalUsername.Buffer, 128, "Administrator");
  RtlInitUnicodeString(&LocalUsername.String, LocalUsername.Buffer);

  RtlCopyMemory(OutputUsername, &LocalUsername, sizeof(USERNAME));
  OutputUsername->String.Buffer = OutputUsername->Buffer;

  return STATUS_SUCCESS;
}
```

Listing 34: Example double-write condition revealing a kernel-mode address

Let's examine the example illustrated in Listing 34. The `USERNAME` structure is a self-contained object that includes both the `UNICODE_STRING` structure and the corresponding textual buffer. A local object of this type is first initialized in lines 11-12 and later copied to the client in line 14. Since the `Buffer` pointer passed back to user-mode still contains a kernel address, it is overwritten with a reference to the ring 3 buffer in line 15. The time window available for another thread to capture the disclosed kernel pointer lasts between lines 14 and 15.

Detection of double writes is again very similar to that of double fetches – the instrumentation should catch all kernel→userland memory writes, and signal a bug every time a specific address is written to with non-zero data more than once in the context of a single system call. As an additional feature, the tool can put a special emphasis on cases where the original bytes resemble a kernel-mode address, and the new data appears to be a user-mode pointer. This should help highlight the reports most likely to represent actual information disclosure bugs.

To test the above idea, we implemented a simple prototype of the instrumentation and ran it on Windows 7 and 10 32-bit. As a result, we discovered three double-write conditions, all leaking addresses of objects in ring 0 memory:

- A bug in `nt!IopQueryNameInternal`, in the copying of a `UNICODE_STRING` structure. The flaw is reachable through the `nt!NtQueryObject` and `nt!NtQueryVirtualMemory` system calls, and was filed in the Project Zero bug tracker with the corresponding proof of concept as issue #1456 [63].
- A bug in `nt!PspCopyAndFixupParameters` (`UNICODE_STRING` structures nested in `RTL_USER_PROCESS_PARAMETERS`).
- A bug in `win32k!NtUserfnINOUTNCCALCSIZE` (`NCCALCSIZE_PARAMS` structure).

The first of the above problems was reported to Microsoft in December 2017, but the vendor replied that the report and all similar issues didn't meet the bar for a security bulletin and would be instead targeted to be fixed in the next version of Windows. Upon publishing the details of the double-write conditions, other researchers publicly claimed that they were also aware of the bug class and collided with some of our findings [28, 30, 29].

A.8 Ignored function return values

Correctly checking the return values of functions and methods is essential to maintaining a consistent program state and preventing unexpected conditions, which can further lead to security flaws. This type of issues was categorized by MITRE and assigned a CWE-252 weakness ID [95]. There are many examples of real-world bugs caused by insufficient validation of return values, for instance a buffer overflow in the FreeType font engine [48], a NULL pointer dereference in `win32k.sys` exploited in the wild [110], or a lesser stack-based infoleak in `win32k.sys` related to unicode strings [47]. In all of the above cases, properly handling the error conditions indicated by the called functions would have prevented the subsequent vulnerabilities.

It is important to note that unchecked return values are a canonical type of a problem that can be effectively detected using static analysis, especially if the source code is available. As a very basic example, after finding the aforementioned FreeType bug caused by an ignored error code, we added a `__attribute__((warn_unused_result))` directive to the declaration of the internal `FT_Error` type and compiled the project with the `-Wno-attributes` flag, which caused gcc to warn about all instances of unchecked return values of type `FT_Error` [49]. The output of this experiment motivated the project's maintainer to submit a series of patches to fix many potential, related bugs. While this is a simple example, more advanced analyzers should be able to pinpoint such behavior accurately, without the need to apply any special changes to the tested code.

On a binary level, static analysis is more difficult, as some information is inevitably lost during compilation. However, it is still possible to achieve by tracking operations on the `EAX` or `RAX` registers to determine if each of the evaluated functions has a return value, and verifying if all of their callers check that value accordingly. A big advantage of the static approach is that it is able to process an entire code base without the need to execute it, and hence it is not limited by the reachable code coverage. Nonetheless, this can also be considered a drawback, as reports regarding unchecked return values tend to be flooded with false-positives and non-issues. In this context, the results of dynamic analysis are easier to triage and understand, because they are supplemented with complete information about the system state, including traces of the control flow, the actual values returned by the functions and so on.

The execution pattern indicating potential problems in the code is straightforward. Excluding minor corner cases and assuming 32-bit execution mode, it is as follows: if two instructions set the value of the `EAX` register and they are separated by at least one `ret` instruction but no reads from `EAX` in between, this suggests that the second write discards a return value that should have been checked first. The only major problem with the above scheme is the fact that Bochs doesn't provide instrumentation callbacks for register operations. On the upside, references to the emulated CPU registers are achieved through general-purpose macros such as `BX_READ_32BIT_REG` and `BX_WRITE_32BIT_REGZ` defined in `cpu/cpu.h` (Listings 35 and 36). For a demonstration of how the macros are used in the software implementation of the `mov r32, r32` instruction, see Listing 37. Thanks to this detail, we were able to introduce two custom callbacks named `bx_instr_genreg_read` and `bx_instr_genreg_write`, invoked on every access to any register; their prototypes are shown in Listing 38. We subsequently added calls to these instrumentation-defined functions in all register-related macros found in `cpu/cpu.h`.

With the capability of intercepting all references to `EAX` taking place in the kernel, we implemented the high-level logic of the instrumentation. While testing the tool, we learned that several instructions required special handling – `xor eax, eax` had to be treated as *write* operations instead of the theoretical *r/w*, while `movzx eax, ax` and similar instructions are effectively no-ops in the sense of our logic, even though they operate on various parts of `EAX`.

```
#define BX_READ_8BIT_REGL(index) (BX_CPU_THIS_PTR gen_reg[index].word.
    byte.rl)
#define BX_READ_16BIT_REG(index) (BX_CPU_THIS_PTR gen_reg[index].word.rx)
#define BX_READ_32BIT_REG(index) (BX_CPU_THIS_PTR gen_reg[index].dword.
    erx)
```

Listing 35: Definitions of macros reading from CPU registers in Bochs (cpu/cpu.h)

```
#define BX_WRITE_32BIT_REGZ(index, val) {\
  BX_CPU_THIS_PTR gen_reg[index].rrx = (Bit32u) val; \
}

#define BX_WRITE_64BIT_REG(index, val) {\
  BX_CPU_THIS_PTR gen_reg[index].rrx = val; \
}
#define BX_CLEAR_64BIT_HIGH(index) {\
  BX_CPU_THIS_PTR gen_reg[index].dword.hrx = 0; \
}
```

Listing 36: Definitions of macros writing to CPU registers in Bochs (cpu/cpu.h)

```
BX_INSF_TYPE BX_CPP_AttrRegparmN(1) BX_CPU_C::MOV_GdEdR(bxInstruction_c *
    i)
{
  BX_WRITE_32BIT_REGZ(i->dst(), BX_READ_32BIT_REG(i->src()));

  BX_NEXT_INSTR(i);
}
```

Listing 37: Implementation of the `mov r32, r32` instruction in Bochs (cpu/data_xfer32.cc)

```
void bx_instr_genreg_read(unsigned cpu, unsigned index, unsigned size,
    bool is_high);

void bx_instr_genreg_write(unsigned cpu, unsigned index, Bit64u value,
    unsigned size, bool is_high);
```

Listing 38: Prototypes of custom register-related instrumentation callbacks

We evaluated the instrumentation against Windows 7 32-bit and collected over 2700 unique reports of unchecked return values. Due to the excessive output volume we were only able to review about 20% of the reports, which did not manifest any high-severity bugs. Nevertheless, we believe the technique shows great potential and can be successfully used to uncover new bugs, with more effort put into reducing the number of flagged non-issues.

A.9 API misuse

Kernel instrumentation is not limited to detecting low-level bugs related to interactions with user-mode memory, but can also prove useful in spotting problems on a higher level, such as the way API functions are called. Because of the complexity of certain kernel subsystems (e.g. the *Object Manager* in Windows [93]), some of the clients may use their interface in unsafe, exploitable ways. Examples of Windows kernel vulnerabilities that could be caused by API misuse are listed below:

- One of the arguments of the `ObReferenceObjectByHandle` function [87] is `ObjectType`, a pointer containing the anticipated type of the referenced handle, or `NULL`, if the object may be of any type. Broadly setting the parameter to `NULL` while still expecting the object to be of a specific type can lead to type confusion and memory corruption.
- Windows kernel objects are subject to reference counting, which makes them potentially prone to typical problems such as *refcount leaks* (more references than dereferences in one self-contained interaction) or *double derefs* (vice versa) [62]. Incorrect balancing of the (de)references usually leads to use-after-free conditions.
- There are two types of Windows object handles – user-mode and kernel-mode ones. It is essential for system security that all handles not explicitly exported to user-mode are created with the `OBJ_KERNEL_HANDLE` flag set in the `OBJECT_ATTRIBUTES.Attributes` field; otherwise, rogue programs could access system objects and abuse them for privilege escalation. Signs of unsafe behavior include creating user-mode handles and not writing their numerical values to ring 3 memory, or creating temporary user-mode handles and destroying them in the scope of the same system call.

All of the above and many other types of API-related issues could be uncovered by instrumenting known sensitive functions and validating their security-related requirements.