Windows Security Hardening Through Kernel Address Protection

Mateusz "j00ru" Jurczyk

August 2011

Abstract

As more defense-in-depth protection schemes like Windows Integrity Control or sandboxing technologies are deployed, threats affecting local system components become a relevant issue in terms of the overall operating system user's security plan. In order to address continuous development of *Elevation of Privileges* exploitation techniques, Microsoft started to enhance the Windows kernel security, by hardening the most sensitive system components, such as Kernel Pools with the *Safe Unlinking* mechanism introduced in Windows 7[19]. At the same time, the system supports numerous both official and undocumented services, providing valuable information regarding the current state of the kernel memory layout. In this paper, we discuss the potential threats and problems concerning unprivileged access to the system address space information. In particular, we also present how subtle information leakages can prove useful in practical attack scenarios. Further in the document, we conclusively provide some suggestions as to how problems related to kernel address information availability can be mitigated, or entirely eliminated.

1 Introduction

Communication between distinct executable modules running at different privilege levels or within separate security domains takes place most of the time, in numerous fields of modern computing. Both hardware- and software-enforced privilege separation mechanisms are designed to control the access to certain resources - grant it to modules with higher rights, while ensuring that unathorized entities are not able to reach the protected data.

The discussed architecture is usually based on setting up a trusted set of modules (further referred to as "the broker") in the privileged area, while having the potentially malicious code (also called "the guest") executed in a controlled environment. In order for the low-integrity programs to retain their original functionality, the broker usually provides a special communication channel, through which the guest can make use of certain services implemented by the broker. While effectively limiting the spectrum of potential action which can be taken by the client, the approach also guarantees that untrusted code can only do as much as the system user or developers really intend to (assuming a flawless implementation of the trusted code). A basic model of the architecture is presented on Figure 1. Operating systems, sandboxing and virtualization technologies all make a good example of computer software taking advantage of privilege separation.

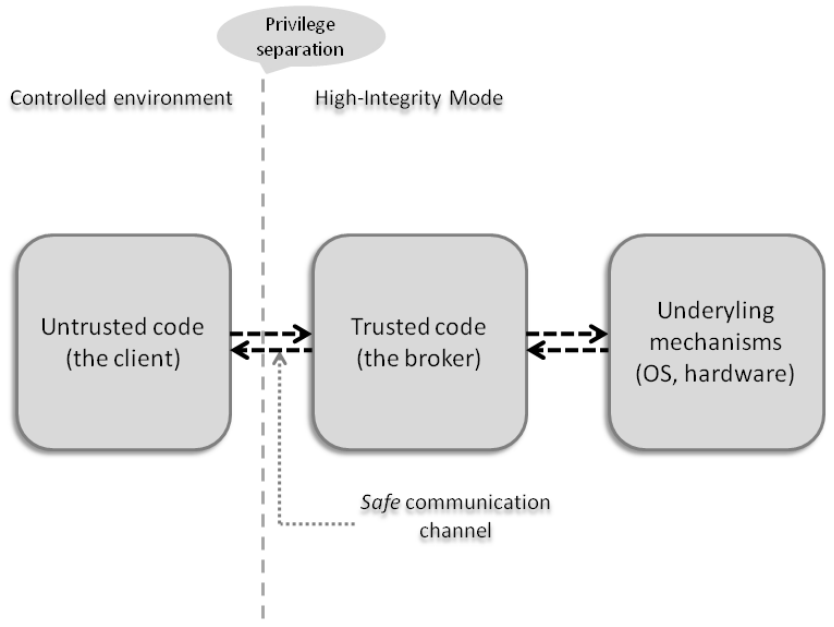
Privilege separation
Controlled environment
High-Integrity Mode
Untrusted code (the client)
Trusted code (the broker)
Underyling mechanisms (OS, hardware)
Safe communication channel

Figure 1: A typical design of a client-broker privilege separation scheme

Brokers, as regular pieces of executable code, do suffer from regular software bugs. As a direct consequence of communicating and processing data received from less-trusted modules, these bugs might often be triggered through a specially crafted *dialogue* with one of the clients. Furthermore, since some of these programming bugs may - and often have - security implications, brokers can be specifically subject to software vulnerabilities. Given the fact that some security flaws can expose a way to accomplish highly-privileged code execution from within an untrusted client, the overall security architecture can be potentially circuvmented by exploiting one security issue in a single trusted module.

Despite the usual methods of reducing the amount of software security problems found in brokers - fuzzing and source code auditing - efforts have been made to address the consequences of software bugs in a more generic way. Namely, Microsoft - as well as other operating system vendors - introduced several anti-exploitation mechanisms, purposed to render some local vulnerabilities completely useless, and make it considerably harder to use others. The most commonly known security features implemented in Windows are: *Stack Cookies*[24], *Heap Protection* [8] (*heap cookies*, *safe unlinking* etc.), *Exception-Handling Protection* [7] (*SafeSEH*, *SEHOP* etc.), *Data Execution Prevention* [21] and *Address Space Layout Randomization* [6]. The above mitigation techniques can be divided into three, general types:

1. Prevention of undesired actions
2. Integrity check of the internal state
3. Randomization of the internal state

The first group of mechanisms is designed to stop every effort made to perform actions, which are otherwise considered undesired or suspicious (e.g. code execution from *non-executable* pages). The second group aims to examine if the program's internal integrity has been damaged, which usually implies that an attack against a security issue is in progress. Both types of mitigations work in a completely deterministic manner, as they only ensure that no potentially harfmul operation are, or were performed on the local machine.

As opposed to the first two groups, the sole purpose of internal state randomization is not to detect vulnerability exploitation in itself, but rather to make the application's execution path dependent on a *random* factor, ideally unable to be guessed by a potential attacker. This very approach is taken by *Address Space Layout Randomization*, which deliberately relocates executable images to random locations, thus making it difficult or impossible to build a reliable exploit by using hard-coded addresses.

Since the security level of a program often relies on how hard it is for the client to predict the broker's internal state, it becomes obvious that the latter should never reveal more information, than actually required for a client to function properly. The desire of memory layout information can be easily observed in the context of web-browsers, where any kind of information leakage to javascript code is considered a legitimate and valuable security vulnerability.

Interestingly, Microsoft does not seem to follow the discussed principle in terms of user- and kernel- mode transitions. The operating system includes specific address leaks as parts of its regular functionality, and even provides documented API interface for some of these services. In our opinion, this quasi-correct behavior is a result of a lack of an official policy as of how important it is to keep kernel addresses secret. In order to mitigate the threats caused by careless ring-0 address management, we conclusively present some steps, which can be taken by Microsoft to further eliminate the disclosure of sensitive addresses, while retaining the old functionality.

The rest of the paper is organized as follows. In Section 2, we review the different types of addresses made available to regular user-mode applications, and what is their specific meaning in the operating system. In Section 3, we discuss the usages of the revealed addresses, in terms of practical kernel exploitation scenarios. In Section 4, we propose ways of reducing the impact implied by memory layout information leakages, as well as possible fixes on both hardware and software level. Finally, in Section 5 we provide thoughts and suggestions on the future of kernel address information availability, and in Section 6 we provide a conclusion of the paper.

2 Address space information sources

In this section, we review the existing means, by which regular programs can obtain information regarding the kernel memory layout.

2.1 Windows System Information classes

Since the very early days of the Windows NT-family system development, the kernel provided a centralized service, which would be used to query any type of information regarding the current system state, from both user- and kernel-mode. This specific system call is named *NtQuerySystemInformation* (see Listing 1), and currently manages more then 80 information classes (specified by the `SYSTEM_INFORMATION_CLASS` enum, defined in `winternl.h`).

Listing 1: NtQuerySystemInformation definition

```
NTSTATUS
STDCALL
NtQuerySystemInformation(
  SYSTEM_INFORMATION_CLASS SystemInformationClass,
  PVOID   SystemInformation,
  ULONG   SystemInformationLength,
  PULONG  ReturnLength);
```

The current amount of possible query types is caused by a legacy policy - once introduced, probablynone of the enumeration members has ever been removed from the service implementation. The avaiable information types include, but are not limited to the following items:

- Basic system and machine characteristics,
- System performance,
- Date / Time,
- State of processes and threads,
- Object Manager information,

The service does not require any specific privileges from the requestor, thus every information class is available to every program running on the operating system. Consequently, the routine makes a great source of utile information, which can be used by a local attacker, previous to performing an *Elevation of Privileges* attack against the machine.

In further subsections, we review the particular information classes, that can be used to obtain a solid amount of kernel memory addressing details. We will briefly characterize the internal structures used to describe the system state, before moving to specific scenarios, in which the obtained information can turn out to be of great value.

2.1.1 SystemModuleInformation

The information class is used to retrieve basic data regarding all device drivers (including core Windows ring-0 modules) presently loaded into kernel space. Should the service succeed, the output buffer contains a list of the `SYSTEM_MODULE_INFORMATION` structures (see Listing 2).

Listing 2: Kernel module descriptor

```
typedef struct _SYSTEM_MODULE_INFORMATION
{
  ULONG Reserved[2];
  PVOID Base;
  ULONG Size;
  ULONG Flags;
  USHORT Index;
  USHORT Unknown;
  USHORT LoadCount;
  USHORT ModuleNameOffset;
  CHAR ImageName[256];
} SYSTEM_MODULE_INFORMATION, *PSYSTEM_MODULE_INFORMATION;
```

Among other items, three structure fields are particularly interesting: `Base`, `Size` and `ImageName`. As their names indicate, these fields represent the image base (`IMAGE_OPTIONAL_HEADER.ImageBase`), size (`IMAGE_OPTIONAL_HEADER.SizeOfImage`) and file name of a single kernel module. In other words, it is possible for any user to create a complete map of device driver memory placement across the privileged address space. An exemplary output snippet of a simple utility, making use of the discussed information class, is presented in Listing 3.

Listing 3: A custom *driverquery* utility output

```
Name:     ntoskrnl.exe, ImageBase: 0x8281c000, ImageSize: 0x003ab000
Name:          hal.dll, ImageBase: 0x82bc7000, ImageSize: 0x00033000
Name:        kdcom.dll, ImageBase: 0x8a809000, ImageSize: 0x00007000
Name:        PSHED.dll, ImageBase: 0x8a810000, ImageSize: 0x00011000
Name:      BOOTVID.dll, ImageBase: 0x8a821000, ImageSize: 0x00008000
Name:         CLFS.SYS, ImageBase: 0x8a829000, ImageSize: 0x00041000
Name:           CI.dll, ImageBase: 0x8a86a000, ImageSize: 0x000e0000
[...]
```

It is important to note that Microsoft created a documented interface around the *SystemModuleInformation* class, and incorporated it into the *Process Status*

API[15]. Namely, the operating system supports the following official routines to examine information about device drivers present in kernel-mode:

- `EnumDeviceDrivers`
- `GetDeviceDriverBaseName`
- `GetDeviceDriverFileName`

Although the kernel-oriented part of *PSAPI* only allows to enumerate drivers' base addresses and names, it is remarkable that Microsoft decided to make a part of the *SystemModuleInformation* functionality available to regular developers; it might potentially have future consequences in terms of legacy, if the vendor starts making efforts towards reducing the kernel address accessibility surface.

2.1.2 SystemHandleInformation

The information class was designed to provide general information about all `HANDLE` values (and the associated objects) from all processes present in the system. On output, the caller receives an array of the `SYSTEM_HANDLE_INFORMATION` structures (see Listing 4), each maintaining data about a single numeric resource ID.

Listing 4: `HANDLE` descriptor

```
typedef struct _SYSTEM_HANDLE_INFORMATION {
  ULONG ProcessId;
  UCHAR ObjectTypeNumber;
  UCHAR Flags;
  USHORT Handle;
  PVOID Object;
  ACCESS_MASK GrantedAccess;
} SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_INFORMATION;
```

The descriptor contains every relevant `HANDLE` characteristic, hence making an invaluable source of information. Most importantly, the kernel provides the requestor with an address of the object body, referenced by the given `HANDLE`. Thanks to the functionality, it becomes possible to enumerate all handles managed by the operating system, including processes with privileges higher than the original information requestor.

One potential problem related to the original `HANDLE` descriptor structure, is the fact that the *Handle* field is declared as `USHORT`, implying 16-bit storage width. Considering that the handle growth incremental on Windows is four, and a single process can potentially own more than 16384 handles, the structure lacks the upper 16 bits of numeric handle representation. In certain scenarios - such as using objects to spray the kernel address space - this issue can render the overall technique useless, by making it impossible to distinguish numeric `HANDLE` values of, for example, 0x0007C and 0x1007C. In order to avoid the problem, we advice to use the *SystemExtendedHandleInformation* class, together with the SYSTEM_HANDLE_INFORMATION_EX structure (see Listing 5).

Listing 5: Extended HANDLE descriptor

```
typedef struct _SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX {
  PVOID Object;
  HANDLE UniqueProcessId;
  HANDLE HandleValue;
  ACCESS_MASK GrantedAccess;
  USHORT CreatorBackTraceIndex;
  USHORT ObjectTypeIndex;
  ULONG HandleAttributes;
  PVOID Reserved;
} SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX, *
    PSYSTEM_HANDLE_TABLE_ENTRY_INFO_EX;
```

In Listing 6, an exemplary output snippet of the *handlequery* utility is presented.

Listing 6: First records of the *SystemExtendedHandleInformation* output

```
[0]: PID: 0x00000004, Handle: 0x00000004, Object: 0x84a43a90
[1]: PID: 0x00000004, Handle: 0x00000008, Object: 0x8bc58158
[2]: PID: 0x00000004, Handle: 0x0000000c, Object: 0x8bc13e68
[3]: PID: 0x00000004, Handle: 0x00000010, Object: 0x8bc11658
[4]: PID: 0x00000004, Handle: 0x00000014, Object: 0x8bc72e38
[...]
```

2.1.3 SystemLockInformation

Upon invoking NtQuerySystemInformation with this information class, the operating system returns a list of lock descriptors, contained in the SYSTEM_LOCK_INFORMATION (see Listing 7). The *locks* are otherwise known as ERESOURCE structures, and are (only) available to kernel-mode, to implement exclusive/shared synchronization. For more information about the mechanism, see *Introduction to ERESOURCE Routines*[17], or specifically, the ExInitializeResourceLite routine documentation.

Listing 7: ERESOURCE descriptor

```
typedef struct _SYSTEM_LOCK_INFORMATION {
  PVOID Address;
  USHORT Type;
  USHORT Reserved1;
  ULONG ExclusiveOwnerThreadId;
  ULONG ActiveCount;
  ULONG ContentionCount;
  ULONG Reserved2[2];
  ULONG NumberOfSharedWaiters;
  ULONG NumberOfExclusiveWaiters;
} SYSTEM_LOCK_INFORMATION, *PSYSTEM_LOCK_INFORMATION;
```

2.1.4 SystemExtendedProcessInformation

On the Windows platform, the OS allocates two distinct stacks for every regular thread: a user- and kernel-mode stack. Intuitively, each of them is used within the corresponding privilege level, thus protecting the more privileged execution flow from any ring-3 disruptions. Although not being able to operate on the kernel stack, user-mode code can obtain its base address and size through the *SystemExtendedProcessInformation* information class. More specifically, the discussed class can be used to retrieve very detailed data regarding all processes and threads running on the system (described by the `SYSTEM_PROCESS_INFORMATION` and `SYSTEM_THREAD_INFORMATION` together with `SYSTEM_EXTENDED_THREAD_INFORMATION` structures, respectively).

Listing 8: Extended thread descriptor

```
typedef struct _SYSTEM_EXTENDED_THREAD_INFORMATION
{
  SYSTEM_THREAD_INFORMATION ThreadInfo;
  PVOID StackBase;
  PVOID StackLimit;
  PVOID Win32StartAddress;
  PVOID TebAddress;
  ULONG Reserved1;
  ULONG Reserved2;
  ULONG Reserved3;
} SYSTEM_EXTENDED_THREAD_INFORMATION, *
    PSYSTEM_EXTENDED_THREAD_INFORMATION;
```

2.2 Win32k.sys Object Handle Addresses

Similarly to the Windows kernel executive, the major graphical device driver - win32k - also manages its own per-session handle table for USER and GDI handles. The table is initialized in `win32k!Win32UserInitialize`, and stored at the base address of a shared section, `win32k!gpvSharedBase`. This section is subsequently mapped into every GUI process running in the system, making it possible for processes to access the handle table without resorting to a system call. Mapping the shared section into user-mode memory areas was considered beneficial in terms of general system effectiveness, efficiently reducing the number of context and privilege switches required to perform graphical operations.

The address of the shared section can be obtained by numerous means, e.g. by scanning all sections mapped in the local memory context, or through an exported `user32!gSharedInfo` symbol (present only on Windows 7).

A single entry in the handle table is represented by a `HANDLENTRY` structure, as shown in Listing 9. Among other fields, the `phead` and `pOwner` members contain the address of the object, and the handle owner (either an `ETHREAD` or `EPROCESS` pointer).

Listing 9: Win32k handle table entry

```
typedef struct _HANDLEENTRY {
  struct _HEAD* phead;
  VOID* pOwner;
  UINT8 bType;
  UINT8 bFlags;
  UINT16 wUniq;
}HANDLEENTRY,*PHANDLEENTRY;
```

As mentioned, it is possible to enumerate the win32k handle table by just operating on the local process memory, without resorting to a single system call (except for the one required to convert the program to a GUI process). Also, as a direct consequence of the shared section scope, one application can list all objects created within the same session. For more information as for how to correctly find and manage the handle table, see *Kernel Attacks Through User-Mode Callbacks* [22].

2.3 Win32k.sys System Call Information Disclosure

As discovered several months prior to writing the paper, more then twenty win32k system call handlers were leaking kernel-mode addresses to user-mode through the return value. As it later turned out, the information disclosure was caused by invalid definitions of the flawed services. Instead of declaring the return value to have the same bit-width as the native processor word (32 or 64, depending on the platform), the definitions of several system calls were similar to those presented in Listing 10 and 11.

Listing 10: Exemplary win32k service with no return value

```
VOID NtUserRandomService( [...] );
```

Listing 11: Exemplary win32k service with a narrow return value type

```
USHORT NtUserRandomService( [...] );
```

Consequently, the compiled routines would either leave the EAX/RAX register (through which return values are passed in `STDCALL`) unitinialized, or only initialize the least significant 16 bits, leaving the remaining part unchanged.

When no value is explicitly returned, the *actual* return value depends on the last EAX/RAX register modification, prior to leaving the system call. Hence, it is potentially possible that such an *Information Disclosure* would reveal stack / heap data or random kernel memory addresses. As further investigation showed, the affected functions' epilogues had usually a very similar format, presented in Listing 12.

Listing 12: A typical epilogue of a win32k system call handler

```
.text:BF853847       call    _LeaveCrit@0
.text:BF85384C       pop     esi
.text:BF85384D       pop     ebp
.text:BF85384E       retn    0Ch
```

The internal `LeaveCrit` function initializes EAX/RAX with the address of the current thread's `ETHREAD` structure. Despite this one type of address, it is also possible to retrieve a pointer to the local `W32THREAD` structure, through five routines with a slightly different epilogue. For more information about the issue, see *Subtle information disclosure in WIN32K.SYS syscall return values*[11].

Even though both kernel-mode addresses revealed through invalid return types can also be obtained by other means, this behavior is strictly *conincidental*, and the operating system developers are very unlikely to have any control over the nature of the disclosed information. Therefore, the current low impact of such subtle leakages might grow up to a serious problem in the future, especially in case of steps being taken to reduce the amount of kernel address space information available to unprivileged applications.

2.4 Descriptor Tables

In this section, we review the types and ways of reaching kernel addresses related to *Descriptor Tables*, a crucial part of the Intel x86 and x86-64 CPU architecure, on the Windows platform.

2.4.1 SIDT, SGDT

Every Intel-architecture processor (or a single core) makes extensive use of three *Descriptor Tables*:

- Interrupt Descriptor Table
- Global Descriptor Table
- Local Descriptor Table

The *Interrupt Descriptor Table* consists of 255 entries, each associating an exception or interrupt vector with a gate descriptor for the procedure or task used to service the associated exception or interrupt.

The *Global Descriptor Table* represents a set of 8-byte entries, each describing a Code Segment, Data Segment, TSS, Call-Gate or LDT. The table is an essential component of *segmentation*, the first step in address translation. It also plays an important role in terms of privilege separation. Both of the discussed structures have a global system scope, and once initialized, they almost never change during Windows run time. *Local Descriptor Table*, on the other hand, is a local equivalent of GDT. It is an optional structure with per-process scope, which can be set up by the kernel on demand [26].

The *Interrupt* and *Global Descriptor Tables* are localized through virtual addresses. These addresses are stored in dedicated registers called `IDTR` and `GDTR`, respectively. Write access to these registers is accomplished through privileged `LIDT` (*Load IDT*) and `LGDT` (*Load GDT*) instructions. Trying to execute one of them within a higher *ring* results in an immediate #GP(0) exception. On the other hand, reading the registers' values is not restricted by any means, and can be achieved through corresponding `SIDT` and `SGDT` instructions. As *Intel 64 and IA-32 Architectures Software Developer's Manual* states [2]:

> SIDT is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated.

In order to retrieve the addresses of Descriptor Tables for all active processors or cores, it is necessary to use the `SetThreadAffinityMask` API. It is also worth to note, that the `SIDT` instruction functionality has already been used in the past to detect the presence of VMM environment, as presented by Joanna Rutkowska in 2004 [5].

2.4.2 GDT Entries

In spite of the *Global Descriptor Table* address availability alone, Windows also allows to obtain and examine particular table entries. The functionality is operable through a documented `GetThreadSelectorEntry` function, which is internally implemented using `NtQueryInformationThread` together with the `ThreadDescriptorTableEntry` information class.

Index	**Type**	**Base**	**Limit**	**DPL**	**Notes**
5	tss	80042000	20AB	0	Task State Segment (per-processor)
6	data	FFDFF000	FFF	0	Windows Processor Control Region (per-processor)
9	ldt	86811000	7	0	optional custom LDT (per-process)

Table 1: Kernel-mode entries in a typical Windows Global Descriptor Table

Since the operating system puts no limitation on the segment selectors being queried or the completness of GDT information, it is possible to scan the overall table, collecting kernel-mode addresses. Table 1 (see a complete version of the table [13]) presents entries containing kernel-mode base addresses. As the table shows, GDT contains a total of three entries, which might prove useful for a potential attacker. The first two are present regardless of the current system state, as they are essential for correct CPU (*Task State Segment*), and system (*Processor Control Region*) performance. As previously mentioned, the third item (9th index) is not initialized by default; it is only created (and remains active throughout the process lifespan) upon creating the first LDT entry with a dedicated service.

3 Exploitation usability

In this section, we focus on certain software vulnerability classes and scenarios, in which each of the disclosed type of address may come in handy.

3.1 SystemModuleInformation class

Free access to information concerning all executable images residing within the boundaries of kernel virtual address space makes it a powerful tool in numerous exploitation contexts. This is primarily caused by the diversity of data types present in a single PE file - executable code, function pointers, static variables, large arrays, exported symbols - each of which represents a certain value, depending on a given vulnerability characteristics.

3.1.1 Pre-exploitation payload initialization

The official user-mode Windows API interface is split into tens of separate libraries, such as `kernel32.dll`, `user32.dll`, and so on, depending on the nature and functionality of a certain function set. As opposed to ring-3, a great majority of documented Windows kernel API is located inside the primary OS core - `ntoskrnl.exe` (or its equivalent); while the other part (such as `KeRaiseIrql`) resides in `HAL.DLL`.

Listing 13: A pseudo-code *GetKernelProcAddress* implementation

```
LPVOID GetKernelProcAddress(PCHAR Module, LPCSTR lpProcName)
{
  HMODULE ModuleHandle;
  FARPROC ProcPointer;

  if((ModuleHandle = LoadLibraryEx(Module, NULL,
      DONT_RESOLVE_DLL_REFERENCES)) == NULL)
    return NULL;

  if((ProcPointer = GetProcAddress(ModuleHandle, lpProcName) == NULL
      )
  {
    FreeLibrary(ModuleHandle);
    return FALSE;
  }

  FreeLibrary(ModuleHandle);

  return (ProcPointer - ModuleHandle + GetDriverImageBase(Module));
}
```

Thanks to such design, it becomes possible to obtain the virtual address of any Windows kernel routine, being part of the documented DDK API. The task can be achieved, by combining the *SystemModuleInformation* functionality with popular image management routines like `LoadLibraryEx` or `GetProcAddress` (see Listing 13).

Since a typical payload would usually take advantage of the kernel API to load an arbitrary driver (`nt!ZwLoadDriver`) or elevate process privileges (`nt!ZwOpenProcessToken`, `nt!ZwDuplicateToken` and other), it is often best to initialize appropriate pointers in the pre-exploitation stage. This way, any accidential failure at this point can be cleanly processed while still on the ring-3 privilege level.

3.1.2 Return-Oriented Programming

Return-Oriented Programming - previously known as *ret2libc* - is a common exploitation technique, capable of circuvmenting the *Data Execution Prevention* mitigation technology in certain scenarios. The method requires a controlled stack (as a consequence of a typical stack buffer overflow, or upon crafting the stack pointer), and relies on a chained execution of tiny assembly code snippets (referred to as *gadgets*, ending with execution-control instructions, such as `RETN`). In most cases, exploits make use of gadgets residing in executable images loaded in the local address space, thus the technique is considered a sophisticated form of *code reuse*.

Taking advantage of techniques such as ROP is usually motivated with lack of control over the vulnerable process address space. On the other hand, *Elevation of Privilege* attacks assume code execution by definition; a malicious user is usually able to operate within a restricted environment (e.g. process, or user account). Therefore, it is possible to entirely control the user-mode address space during kernel exploitation. Provided that the affected kernel routine executes in the same context as its ring-3 trigger, payload can be successfully executed without the need to control privileged memory areas.

Interestingly, in May 2011 Intel announced a new anti-exploitation technology called *Supervisor Mode Execution Protection*, implemented on the CPU level [3] [4]. The general concept of the upcoming feature is to refuse ring-0 execution of code located in memory pages marked as accessible from user-mode, upon setting the 20^{th} bit in the `CR4` register. Security researchers have already presented possible ways of subverting the protection on both Linux [1] and Windows [14] platforms.

When code execution from user-mode memory is rendered impossible, *Return Oriented Programming* might turn out to become a feasible way of exploiting local Windows kernel vulnerabilities. Should it happen, the ability to retrieve base addresses of PE images present in ring-0 would be a crucial part of the exploitation process. What is even more, as long as the device driver layout is available to untrusted entities, no anti-exploitation measure can stop the attacker from taking over the machine, once the kernel stack is controlled.

3.1.3 Static function pointers

Amongst other classes of kernel security flaws, the *Write-What-Where* condition is one of the most common, and easiest to take advantage of. It can occur as a direct consequence of insufficient input pointer validation, an implicit result of a pool-based buffer overflow, and in several other circumstances (e.g. referencing pointers from the NULL memory page). As the name indicates, the condition allows unprivileged code to *write* a controlled value (*what*) into user-controlled kernel address (*where*).

In most scenarios, the condition can be observed in a four-byte (or eight for Intel x86-64) form, i.e. it is possible to write an operand sized the same as the native CPU word. In order to turn the condition into privileged code execution, it is necessary to overwrite a value, which (directly or implicitly) affects the kernel execution path. Extensive research has been performed in this field [25] [20], resulting in the invention of several effective ideas. One of the most widely known technique, is to overwrite a function pointer, located at a constant offset relative to an exported kernel symbol: `nt!HalDispatchTable + sizeof(ULONG_PTR)`. Upon replacing the original value with the payload virtual address, privileged code execution can be then triggered through the `NtQueryIntervalProfile` service, which invokes the following call stack:

- `nt!NtQueryIntervalProfile`
- `nt!KeQueryIntervalProfile`
- `[nt!HalDispatchTable + sizeof(ULONG_PTR)]`

In general, device driver images contain a tremendous amount of critical spots (such as function pointers), which can be used to compromise a machine through vulnerable kernel code and a *Write-What-Where* condition, including optimized *switch* branch tables, static function pointers or dispatch tables. For as long as device drivers' image bases are not protected from unathorized access, the exploitation of a majority of ring-0 security flaws will remain trivial.

3.2 SystemHandleInformation class

The availability of information about objects with assigned *numeric identifiers* (handles) makes a great source of data regarding the current operating system state. Furthermore, due to the nature and complexity of some of the object types, it is often feasible to use them as a direct post-exploitation stager.

3.2.1 Write-What-Where condition

Similarly to executable modules, some object structures abound in *execution-critical* fields, which might be picked out during a *Write-What-Where* condition exploitation. A list of potential object types includes Timers (`KTIMER`), Threads (`KTHREAD`), or APC Reserve Objects (`KAPC` structure, Listing 14) [12].

Listing 14: Asynchronous Procedure Call descriptor

```
nt!_KAPC
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 Spare0           : Uint4B
   +0x008 Thread           : Ptr32 _KTHREAD
   +0x00c ApcListEntry     : _LIST_ENTRY
   +0x014 KernelRoutine :  Ptr32 void
   +0x018 RundownRoutine   : Ptr32     void
   +0x01c NormalRoutine    : Ptr32     void
   +0x020 NormalContext    : Ptr32 Void
   +0x024 SystemArgument1  : Ptr32 Void
   +0x028 SystemArgument2  : Ptr32 Void
   +0x02c ApcStateIndex    : Char
   +0x02d ApcMode          : Char
   +0x02e Inserted         : UChar
```

3.2.2 Payload storage

Depending on the object design and purpose, user-mode applications may have a varying degree of control over the object's structure contents. Remarkably, several objects (such as `APC Reserve Objects` [10]) allow as much as sixteen bytes of controlled memory placed within the object body. Because of the unlimited access to object address information, it is potentially possible to use the objects as an effective kernel-mode payload container (see Figure 2).

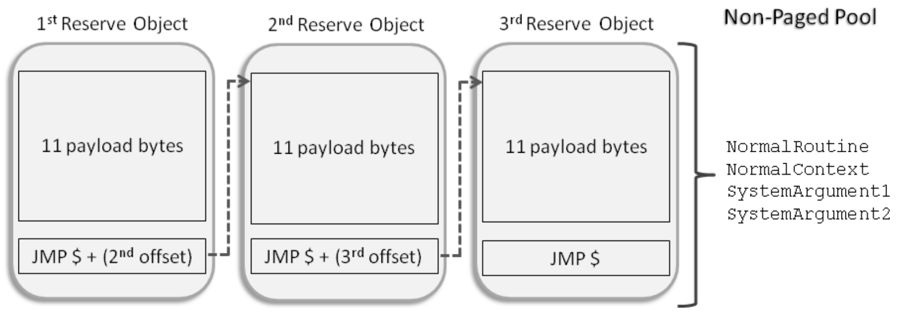

Figure 2: Exemplary `KAPC` structure chain, storing 33 bytes of payload in three chunks of data

One disadvantage of the proposed technique is the fact that the user-controlled shellcode is located inside the kernel pool areas, which are marked as *non-executable*. Fortunately, APC Reserve Objects (as well as a majority of Windows objects) are allocated from *Non-Paged Pool* which - according to MSDN [16] - is excluded from the DEP protection layer.

DEP is also applied to drivers in kernel mode. DEP for memory regions in kernel mode cannot be selectively enabled or disabled. On 32-bit versions of Windows, DEP is applied to the stack by default. This differs from kernel-mode DEP on 64-bit versions of Windows, where the stack, paged pool, and session pool have DEP applied.

3.2.3 Kernel Pool Feng Shui

Analogically to other types of computer software (e.g. web browsers) allowing attacker-controlled code execution (e.g. javascript) and partial control over the internal state of the broker's memory allocator (user-mode heap), the Windows kernel also makes it possible for unprivileged application to affect the pools' (an equivalent of ring-3 heaps) layout. This particular capability can prove especially useful, when dealing with the *Use-after-free* vulnerability class. Moreover, crafting a specific allocations' layout has also been shown to come in handy, in terms of circuvmenting new kernel security features introduced in Windows 7, such as *Safe-Unlinking* [18] [23].

Although little research has been performed in the field of precise kernel pools control, we believe that the subject will become an important point of security researchers' interest, as new anti-exploitation technologies are introduced in the Windows kernel.

3.3 SystemLockInformation

No attacks or exploitation techniques related to the `ERESOURCE` structure addresses are publicly known. Because of the fact that the *Lock* synchronization mechanism is only operable from ring-0, we consider the information class hardly applicable in the context of *Elevation of Privileges* attacks.

3.4 Kernel-mode stacks

As the kernel-mode stack is a crucial part of the ring-0 execution path, it is a perfect candidate for a *Write-What-Where* condition target. Having structures like `KTRAP_FRAME` or stack frames located within a given thread's privileged stack, it is possible to *hijack* ring-0 execution by overwriting a *return-address*, saved `CS:` register within the trap frame, or other sensitive data.

Additionally, the kernel stack can play the role of a data container [9]. This is made possible thanks to the fact, that several Windows services allow an extensive amount of user-mode bytes to be moved to a local stack buffer (up to 4096 bytes). Since all kernel stacks are allocated from non-pageable memory, the above behavior can be made use of in most scenarios scenario, regardless of the vulnerable code IRQL.

Generally speaking, the availability of kernel-mode stack bases is not only useful in terms of generic exploitation, but also turns out to be of great value while evaluating more peculiar types of issues, which are highly dependent on the stack itself (e.g. uninitialized local variable dereferences).

3.5 Win32k.sys shared sections

By mapping the `win32k.sys` shared section into user-mode, the graphical module makes all (session-wide) user / gdi objects' addresses visible to regular applications. Although the information leakage doesn't have any direct security implications, a recent research on a new `win32k.sys` vulnerability vector – user-mode callbacks – has shown that it can heavily simplify the exploitation of the *Use-after-free* vulnerability class.

Since the type of information revealed by the memory mapping is analogous to the *SystemHandleInformation* class, these two information disclosure cases represent a similar degree of usability. The objects managed by kernel-mode Windows subsystem are also believed to be applicable in *Write-What-Where* and *Kernel memory spraying* attacks. No specific advantages of using `win32k.sys` mechanisms instead of the Windows kernel ones are known to the authors.

3.6 Win32k.sys return values

At the time of writing this paper, it is possible to only obtain the addresses of two structures, assigned to the current thread: `KTHREAD` and `W32THREAD`. The first one is equivalent to the current thread's object address, which can be read using *SystemHandleInformation*; the second one is accessible through the local *Thread Environment Block* structure. The possible applications of a *thread* object are discussed in the adequate section, while considering the fact that the latter structure is undocumented and hardly explored, we believe it to be unsuitable for kernel exploitation purposes.

3.7 IDT, GDT and LDT

All of the three primary *Descriptor Tables* consist of bytes representing virtual addresses and privilege level indicators. Consequently, they make an excellent target for *Write-What-Where* attacks. Several ways of altering the `GDT` and `LDT` structures have been described in the *GDT and LDT in Windows kernel exploitation* paper [13], while a number of other ways of poking with the Protected Mode tables are believed to exist.

3.7.1 TSS, PCR

As a direct outcome of the fact that particular GDT entries can be queried by user-mode code, they could be also potentially used to elevate the user's privilege level. In regard to TSS, vulnerable ring-0 code could be used to overwrite parts of the CPU context (such as `SegCs` or `EFLAGS.IOPL`), or modify the I/O Access Bit Mask in such a way, that direct I/O communication with the machine components are available from within user-mode.

4 Mitigations

In this section, we evaluate ways of mitigating the threats incurred by information disclosure issues described in Section 2.

4.1 Windows Kernel Information Classes

Due to the fact that the `NtQuerySystemInformation` classes, revealing kernel address space information, were implemented purposedly and as a feature, they cannot be thought of as regular vulnerabilities. In order to reduce the potential security impact of their functionality, we propose four solutions which we believe might be successfully adopted by Microsoft.

1. Sustain the overall information classes' functionality, except for filling the individual fields, containing kernel-mode information. The concept could be implemented relatively easily in the context of undocumented services; however, taking such a step would also render the device driver-oriented part of the *PSAPI* interface useless, as it is mostly based on image base addresses.

2. Restrict the access to certain information classes, by introducing an additional check on the current process security token (e.g. a `SeTcbPrivilege` requirement). Consequently, only programs with administrative rights would be allowed to query for sensitive kernel information, while making it impossible for restricted users to obtain any of the *classified* data.

   One major disadvantage of the method is the fact that even though it would successfully decrease the amount of data during an *Elevation of Privileges* attack, it would still be feasible to "attack" a 64-bit kernel as a privileged user (administrator), in order to load an unsigned driver into kernel space (a.k.a. admin-to-kernel escalation, or *Driver Signature Enforcement* bypass).

3. Similarly, limit access to sensitive classes by ensuring that only ring-0 callers (i.e. kernel modules) can obtain information regarding kernel addresses. The task could be successfully accomplished by examining the `PreviousMode` value, which represents the `CPL` of code running previous to the `NtQuerySystemInformation` system call.

   The solution is equivalent to the first concept on 32-bit platforms, where administrative privileges imply the ability to load arbitrary device drivers. As previously mentioned, the situation is roughly different on the 64-bit Windows editions, where only digitally signed modules can be loaded into kernel space, unless another option was chosen during the system boot process. Therefore, the discussed method can be considered even more restrictive than the previous one.

4. Entirely cut out the unsafe functionality from the system information service, returning `STATUS_NOT_IMPLEMENTED` in response to any of the four requests regarding kernel address space information.

All of the above suggestions assume more restrictive requirements concerning the availability of several types of information. Although each of them would result in the desired effect, the real problem is *legacy* and *cross-system compatibility*. Formally, Microsoft could modify the behavior of internal, undocumented classes as long as it would not interfere with the vendor's user-mode applications. As it turns out, however, it is very likely that some third-party Windows applications would cease to work after applying the proposed security enhancements. This, in turn, might cause problems much more serious that the benefits of a potentially increased kernel security level.

What is more, one of the blamed information classes - `SystemModuleInformation` - is currently utilized as a part of a documented interface, called *PSAPI*. This fact makes it even harder for the OS developers to make any move, since meddling in an official, established system interface is not a desirable spot. All things considered, we believe that the real future of the awkward system information service will depend on the application compatibility extent Microsoft is willing to give away in lieu of kernel address space secrecy.

4.2 Win32k.sys Handle Table information

As mentioned before, the address space information leakage related to `win32k.sys` is a direct consequence of the current Windows USER/GDI implementation, and numerous efficiency optimizations present therein. The only possible way of obstructing access to graphical objects' address information would be to entirely re-design the current windowing architecture, and implement parts of several crucial system modules (`user32.dll`, `win32k.sys`) from the very beginning. Because of the complexity and potential difficulties related to such an operation, it is highly unlikely that Microsoft will take such a step in existing Windows editions. However, we believe that it might be feasible to apply several major improvements to the current graphical design in the upcoming Windows platforms, as it suffers from other severe architectural problems and issues, as presented by Tarjei Mandt [22].

4.3 Interrupt and Global Descriptor Table

The information disclosure accessible through the `SIDT` and `SGDT` instructions is entirely a matter of the CPU architecture, and is completely unrelated to the operating system intricacies (e.g. it works the same way on the Windows and Linux platforms). Accordingly, the problem must be dealt with on the hardware level.

Although very intuitive and presumably easy to implement, moving the two discussed instructions into the *privileged* group would probably not be the best option, due to *legacy* reasons. Instead, we believe that the CPU would ideally leave the decision of whether `SIDT` and `SGDT` should be available to user-mode or not, to the operating system itself.

A similar approach was taken in terms of protecting ring-0 execution flow from being redirected into user-mode memory pages. Namely, Intel has added a new bit called *SMEP-enable* in the `CR4` register (only controllable by the operating system). Upon setting the flag, the execution of `CPL=0` from memory write-able from `CPL=3` causes an exception to be generated.

> If CR4.SMEP = 1, instructions may be fetched from any linear address with a valid translation for which the U/S flag (bit 2) is 0 in at least one of the paging-structure entries controlling the translation.

We suggest adding an analogous flag, controlling the availability of `IDT` and `GDT` addresses in user-mode, into the `CR4` register. Such a bit (e.g. `CR4.DTAP`, as in *Descriptor Table Address Protection*) would prevent non-privileged code from obtaining the tables' addresses, when set; the original CPU behavior would not be affected otherwise. Such a solution would allow the system developers to assess the benefits and risks related to the *Descriptor Tables'* address accessibility, and choose a corresponding setting. Currently, we are not aware of any measures, which could be taken by Microsoft to address the problem on a software level.

4.3.1 GDT entries

At the time of writing this paper, all `GDT` and `LDT` items can be retrieved by every application, through the `GetThreadSelectorEntry` API. As the function's documentation (*Remarks* section) states:

> Debuggers use this function to convert segment-relative addresses to linear virtual addresses. The ReadProcessMemory and WriteProcessMemory functions use linear virtual addresses.

the function was primarily designed to translate segment selectors into their base addresses, which would make it possible for a debugger to operate on linear, virtual addresses. Due to the fact that user-mode code can only reference memory within user-mode addressing boundaries, the real API's functionality ends on ring-3 segments (most often, the `FS:` segment translation, being the only standard segment with base address other than zero).

Considering the circumstances, we advice that the allowed output of the API function should be reduced to entries with the `DPL` field set to 3. In a more general scenario, the `ThreadDescriptorTableEntry` information class (which `GetThreadSelectorEntry` is based on) would be re-implemented so that it is allowed to only return entries, whose `DPL` is greater or equal to the previous mode `CPL`. Thanks to such approach, device drivers would still be able to make use of kernel segment information, while preventing regular applications from requesting the data.

4.4 Miscellaneous address leaks

In spite of kernel address space information available through documented and undocumented services or certain system architecture characteristics, unintended leakages of information have also been shown to occur. Although eliminating all *information disclosure* issues is very unlikely, we are certain that operating system vendors, or specifically Microsoft, need to take such bugs seriously and fix them accordingly.

5 Remarks

As we have shown in the paper, Windows kernel address space information is an invaluable source of data, making the exploitation of ring-0 vulnerabilities reliable, and relatively easy for a potential, local attacker. This kind of information can be obtained in a variety of ways, starting from well-documented WinAPI routines, up to incomplete return values of graphical system calls. We believe that the current situation is primarily caused by the fact that Microsoft has not had any official policy concerning the disclosure of kernel-related information for years of the system development. Since local kernel attacks were rarely observed and reported in the past, there was no need to establish any rules on what and where kernel addresses could be passed down to user-mode. As ring-0 security is continuously gaining more attention from the security professionals, it now becomes important to state the formal rules, and fix the numerous architectural errors introduced thorough the years of Windows NT development. Although it is clear that kernel address space protection is not an ultimate remedy to successful exploitation of device driver vulnerabilities, it is a big step in terms of overall system security reformation. Remarkably, the Linux kernel developers seem to follow the basic rules of kernel address space information security, as *Information Disclosure* vulnerabilities of that kind are usually treated seriously, and patched within a reasonable period of time.

6 Conclusion

In this paper, we have discussed the many ways of retrieving kernel space addresses of various, internal system components. Furthermore, we have shown how to practically implement each of the presented techniques, and how most of them can be successfully employed during a local *Elevation of Privileges* attack. In order to make it significantly harder to make of kernel-mode security flaws, we conclusively suggested several mitigation techniques and concepts, which could be used to reduce the address space information surface.

References

[1] Dan Rosenberg: *SMEP: What is It, and How to Beat It on*

Linux. http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/.

[2] Intel: *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*. Intel Corporation, 2007.

[3] Intel: *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, 2.5 Control Registers*. Intel Corporation, 2011.

[4] Joanna Rutkowska: *From Slides to Silicon in 3 years!* http://theinvisiblethings.blogspot.com/2011/06/from-slides-to-silicon-in-3-years.html.

[5] Joanna Rutkowska: *Red Pill... or how to detect VMM using (almost) one CPU instruction*. http://invisiblethings.org/papers/redpill.html.

[6] Matt Miller, MSEC Security Science: *On the effectiveness of DEP and ASLR*. http://blogs.technet.com/b/srd/archive/2010/12/08/on-the-effectiveness-of-dep-and-aslr.aspx.

[7] Matt Miller, MSEC Security Science: *Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP*. http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx.

[8] Matt Miller, MSEC Security Science: *Preventing the exploitation of user mode heap corruption vulnerabilities*. http://blogs.technet.com/b/srd/archive/2009/08/04/preventing-the-exploitation-of-user-mode-heap-corruption-vulnerabilities.aspx.

[9] Matthew "j00ru" Jurczyk: *nt!NtMapUserPhysicalPages and Kernel Stack-Spraying Techniques*. http://j00ru.vexillium.org/?p=769.

[10] Matthew "j00ru" Jurczyk: *Reserve Objects in Windows 7*. http://magazine.hackinthebox.org/issues/HITB-Ezine-Issue-003.pdf.

[11] Matthew "j00ru" Jurczyk: *Subtle information disclosure in WIN32K.SYS syscall return values*. http://j00ru.vexillium.org/?p=762.

[12] Matthew "j00ru" Jurczyk: *Windows Objects in Kernel Vulnerability Exploitation*. http://magazine.hackinthebox.org/issues/HITB-Ezine-Issue-002.pdf.

[13] Matthew "j00ru" Jurczyk, Gynvael Coldwind: *GDT and LDT in Windows kernel vulnerability exploitation*. http://vexillium.org/dl.php?call_gate_exploitation.pdf.

[14] Matthew "j00ru" Jurczyk, Gynvael Coldwind: *SMEP: What is it, and how to beat it on Windows.* http://j00ru.vexillium.org/?p=783.

[15] MSDN: *Process Status API.* http://msdn.microsoft.com/en-us/library/ms684884%28v=VS.85%29.aspx.

[16] MSDN: *Windows Objects in Kernel Vulnerability Exploitation.* http://magazine.hackinthebox.org/issues/HITB-Ezine-Issue-002.pdf.

[17] MSDN Library: *Introduction to ERESOURCE Routines.* http://msdn.microsoft.com/en-us/library/ff548046%28v=vs.85%29.aspx.

[18] Peter Beck: *Safe Unlinking in the Kernel Pool.* http://blogs.technet.com/b/srd/archive/2009/05/26/safe-unlinking-in-the-kernel-pool.aspx.

[19] Peter Beck, MSEC Security Science: *Safe Unlinking in the Kernel Pool.* http://blogs.technet.com/b/srd/archive/2009/05/26/safe-unlinking-in-the-kernel-pool.aspx.

[20] Ruben Santamarta: *Exploiting Common Flaws in Drivers.* http://reversemode.com/components/com_remository/com_remository_startdown.php?id=51&chk=52afbf0dc821af727c94451e57f03aab&userid=0.

[21] skape, Skywing: *Bypassing Windows Hardware-enforced DEP.* http://www.uninformed.org/?v=2&a=4&t=pdf.

[22] Tarjei Mandt: *Kernel Attacks Through User-Mode Callbacks.* http://mista.nu/research/mandt-win32k-slides.pdf.

[23] Tarjei Mandt: *Kernel Pool Exploitation on Windows 7.* http://www.mista.nu/research/MANDT-kernelpool-PAPER.pdf.

[24] Tim Burrell, MSEC Security Science: *GS cookie protection effectiveness and limitations.* http://blogs.technet.com/b/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx.

[25] Yuriy Bulygin: *Remote and Local Exploitation of Network Drivers.* http://www.blackhat.com/presentations/bh-usa-07/Bulygin/Whitepaper/bh-usa-07-bulygin-WP.pdf.

[26] Z0mbie: *Adding LDT entries in Win2K.* http://vxheavens.com/lib/vzo13.html.