GDT and LDT in Windows kernel vulnerability exploitation

Matthew "j00ru" Jurczyk and Gynvael Coldwind, Hispasec 16 January 2010

Abstract

This paper describes some possible ways of exploiting kernel-mode *write-what-where* vulnerabilities in a stable manner, on Microsoft Windows NT-family systems. The techniques presented in this document are mostly based on altering processor-specific structures (such as the *Global Descriptor Table* and *Local Descriptor Table*).

This publication does not cover any revolutionary 0-day bugs, nor does it characterize any new vulnerability class. However, it aims to show some interesting methods that could be employed by exploit developers, in order to gain more stability in comparison to existing techniques. Moreover, all hardware-specific information included in this paper is only confirmed to be valid on 32-bit x86 platforms.

The need of a stable exploit path

Nowadays, as kernel-mode vulnerabilities are gaining more and more attention world-wide (under 5% of Microsoft security bulletins in 2007, over 10% in 2008), so are ring-0 exploitation techniques. One of the most common vulnerability class observed over the last couple of years is the *write-what-where condition*, most often a result of improper pointer validation performed by kernel modules (device drivers) or unsafe *pool unlinking*, after triggering a successful buffer overflow.

As the name itself implies, such a vulnerability makes it possible for an attacker to elevate the privileges by performing at least a single WRITE operation in some memory regions that are normally protected from being written to by unprivileged applications (using paging and other memory protection mechanisms). Before conducting a successful attack, however, one must firstly know what are the desired "*what*" and "*where*" operands.

Many publications have already been released, pointing out a number of locations that might be used to gain specific benefits (such as escalating the execution privileges). A list of these papers includes *Exploiting Common Flaws in Drivers*[4], *Exploiting Windows Device Drivers*[3], and *How to exploit Windows kernel memory pool*[6].

The actual problem with the majority of the techniques presented in the above publications is the fact that they are mostly based on undocumented, internal Windows behavior, that is not guaranteed to remain in the same form across system updates, service packs or respective system versions. Therefore, before performing the real exploitation, one is often obliged to check the system version or employ additional heuristic algorithms (e.g. signature kernel memory scanning).

As it turns out, it is possible (and very convenient) to take advantage of some hardware-specific structures, that are undoubtedly implemented across every single Windows version, in the very same way. To be more precise, the structures covered in this paper are the *Global* and *Local Descriptor Tables*, one of the most essential parts of protected mode itself. By overwriting these structures in a certain way, one is able to perform a privilege escalation attack, compatible with every Windows NT system, provided it is running on an x86 processor.

Windows GDT and LDT

The *Global* and *Local Descriptor Tables* are sets of specific x86 structures used to define memory segments for both data and code, as well as other system descriptors like Task State Segments or Call-Gates. Also, the LDT itself is defined as a single GDT entry. For more details regarding the GDT and LDT, please refer to the Intel Manual, Volume 3A[2].

Despite the fact that Microsoft Windows implements a flat memory model, and that protected mode enforces the system to set up at least one Task State Segment per processor, the GDT has more entries than necessary. The table normally consists of a pair of kernel-mode code and data segment entries with *Descriptor Privilege Level* set to 0, a pair of user-mode code and data segment entries (DPL=3), one TSS entry, as well as three additional data segment entries and an optional LDT entry. Neither Windows GDT, nor LDT have any Call-Gate entries, by default. Ta-

Index	Type	Base	Limit	DPL	Notes
0	null				null segment
1	code	0	FFFF (*4KB)	0	kernel-mode code segment
2	data	0	FFFF (*4KB)	0	kernel-mode data segment
3	code	0	FFFF (*4KB)	3	user-mode code segment
4	data	0	FFFF (*4KB)	3	user-mode data segment
5	tss	80042000	20AB	0	Task State Segment (per-processor)
6	data	FFDFF000	1 (*4KB)	0	Windows Processor Control Region (per-processor)
7	data	7FFDF000	FFF	3	Windows Thread Environment Block (TEB) (per-thread)
8	data	400	FFFF	3	used by Windows NTVDM
9	ldt	86811000	7	0	optional custom LDT (per-process)

Table 1: A typical Global Descriptor Table on Microsoft Windows

ble 1 presents the original GDT table of the Microsoft Windows XP system.

The GDT itself, as well as the TSS and PCR descriptor entries it contains, are defined per-processor. The TEB segment is defined per-thread (it is replaced during a thread-context switch), and the optional custom LDT is defined per-process.

By default, a new process does not have any *Local Descriptor Table* defined. Instead, it can be allocated on demand, i.e. when the process requests a new LDT entry to be created (as described by Z0mbie[7]). A pointer to the per-process LDT table is kept inside the associated KPROCESS structure, in the *LdtDescriptor* field. On a context switch to a different process, this field it loaded into the GDT table, and a LGDT command is issued.

The GDT and LDT management functions are scattered between the Win32API and the Native API. These APIs provide the following functions:

- **GetThreadSelectorEntry** – query GDT/LDT entries (Win32API),
- **NtQueryInformationProcess** (*ProcessLdtInformation*) – query LDT entries (Native API),
- **NtSetLdtEntries** – write LDT entries (Native API),
- **NtSetInformationProcess** (*ProcessLdtInformation*) – write LDT entries (Native API),
- other

However, there are several restrictions regarding custom LDT entries, including the base and base+limit being limited to user-space, the segment being accessible in user mode (Descriptor Privilege Level field set to 3), and the segment type being limited to data and code segments only.

Creating a Call-Gate entry in LDT

As we already have some basic knowledge regarding the GDT and LDT structures, it's time to apply real kernel modifications. Our main purpose is to cause the payload to be executed within ring-0 privileges. Therefore, one of the most intuitive ways of transitioning the code flow ring3→ring0 using descriptor tables is probably to set up a Call-Gate, pointing at our own code. Since this mechanism is designed so that it is possible to make such transitions, the only thing we have to do is to make an existing descriptor entry become a Call-Gate, or build one from scratch. Let's start from the beginning, though.

As stated in the previous section, Windows processes don't have any default LDT entry specified. The kernel is responsible for allocating necessary memory the first time the process tries to set up its own local descriptor. Building the LDT in the context of a local process can be achieved by using either the NtSetInformationProcess or NtSetLdtEntries functions (both undocumented). In 2002, Z0mbie implemented and published some exemple functions for playing with the GDT/LDT tables on Windows 2000; a majority of these methods haven't changed since then, though.

Before trying to modify the LDT contents, one should firstly note a few important facts. To begin with, there are multiple consequences of the first **NtSetInformationProcess**(*ProcessLdtInformation*) function call. Primarily, the amount of virtual memory required to hold a new entry is allocated. Next, a special descriptor of the TSS/LDT type is formed and set up in the global table.

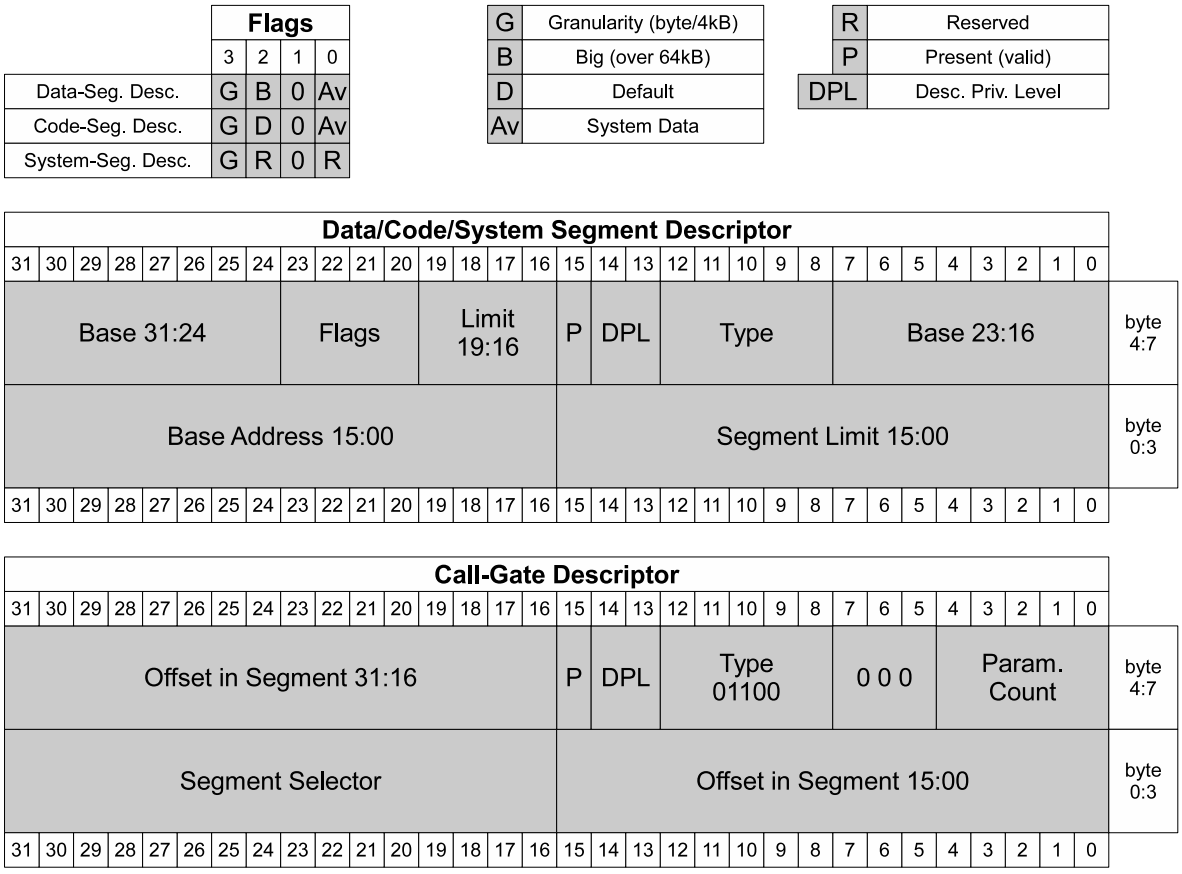

Figure 1: Data Segment (source) compared to the Call-Gate Descriptor (destination)

Because of the fact that the number of *Global Descriptor Tables* is the same as the number of processors the system is running on and, on the other hand, each process may have its own LDT, the system kernel must somehow manage variable GDT entries. In order to do so, every single process has its own *LdtDescriptor* structure inside the KPROCESS object. When a task switch occurs, the thread manager replaces the current LDT descriptor with the one associated with the thread to be executed. Moreover, after the GDT is correctly filled with new data, the IDTR register value is set to a valid GDT index.

After having the LDT created in the context of our process (the contents are not important yet), we need to retrieve the actual table address; two stages are required to achieve this:

1. Get the IDTR register value (segment selector),
2. Get the GDT descriptor, based on the selector from previous step.

The first stage can be easily achieved by using the standard SLDT instruction[5]. As for the second one, Microsoft provides a special API function that can be used to retrieve information about a specified selector – *GetThreadSelectorEntry*[1].

Since Windows does not want any user to be able to create every kind of a descriptor inside LDT, the *NtSetInformationProcess*/*SetLdtEntries* functions are limited to defining only Code/Data (both 16- and 32-bit) segments with DPL=3. Therefore, there are three options available for us:

- Transforming an existing Data descriptor into a Call-Gate,
- Transforming an existing Code descriptor into a Call-Gate,
- Creating a Call-Gate descriptor from the very beginning (using the free area inside LDT).

It is confirmed that all of these options work in practice; however, only the first one is going to be thoroughly described here.

First of all, let's take a look at the two structures we are going to deal with – Figure 1 presents a complete comparison.

Because of the fact that the most commonly observed *write-what-where* vulnerabilities make it possible to overwrite four kernel memory bytes (a single

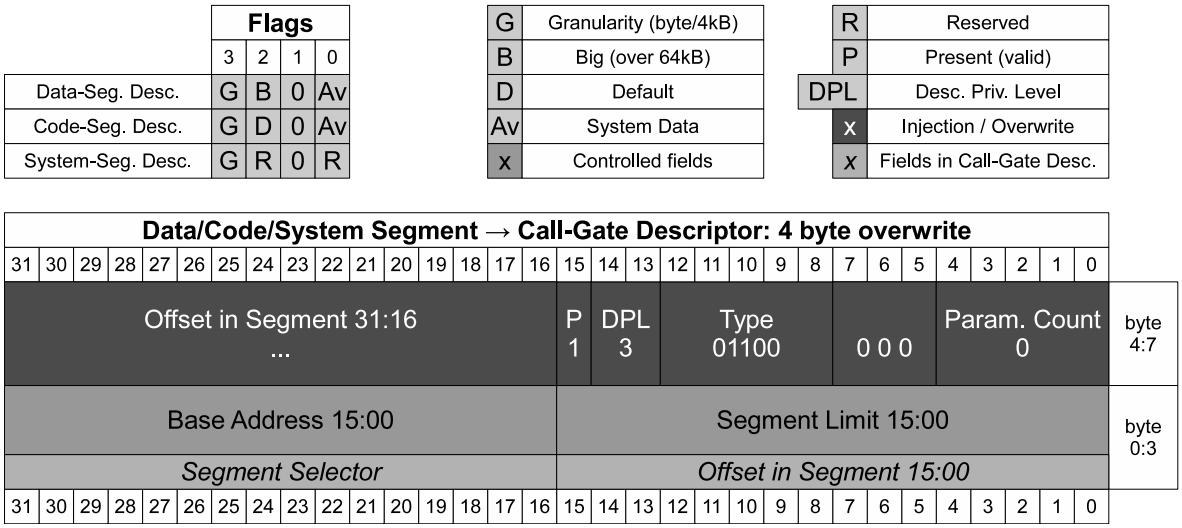

Figure 2: Data and Call-Gate Descriptors compared (four bytes controlled)

machine word), it is important to fit in this limitation. Furthermore, as will be shown, it is also possible to achieve the desired result by modifying only one single byte.

4-byte write-what-where exploitation

Let's begin with the easier variant – modifying 4 bytes out of 8 (which is the actual size of one GDT entry). Since at the time of performing a write operation, the four remaining bytes have to be already properly initialized (i.e. be valid Call-Gate structure fields), the exact structure offset to be written to must be chosen very carefully. In particular, our goal is to control as much of the structure contents as possible – let's take a look at the Data-Segment Descriptor, then. As we're setting up the LDT entry by ourselves, the fields under control are:

- Base Address (*WORD BaseLow, BYTE BaseMid, BYTE BaseHi*) – these three fields, spread across the structure, make up a 32-bit base address of the segment. The only limitation here is the fact that this address must point to somewhere inside user memory (must not be greater than 0x7FFFF000).
- Segment Limit (*WORD LimitLow, DWORD LimitHi:4*) – a 20-bit long value, storing information about how long (in bytes or 4 kilobytes) the specified segment is.

The above two numbers occupy 52 out of 64 bits, which gives us a decent control over the structure. To make things as easy as possible, the WRITE operation should target the second DWORD, containing sensitive information such as Entry Type, DPL and a few additional flags. As mentioned before, we must ensure that the first four bytes are correctly initialized (ready to become a part of the Call-Gate structure). Below, you can find the fields involved in the "structure switch", and what they should be filled with:

- *Segment Limit 15:00* → *Offset in Segment 15:00*: In a Call-Gate, the destination field contains the lower WORD of the destination routine address, which must be passed as the LimitLow field for a Data Segment.
- *Base Address 15:00* → *Segment Selector*: In a call-gate, the segment selector contains the value of CS: register after the privilege mode switch occurs. In practice, it is most likely to be set to 8 (which is the second GDT entry, after the default, empty one).

Having half of the structure already initialized, let's focus on the second part (entirely overwritten while exploiting the *write-what-where condition*). Table 2 shows the proper way of filling the upper 32 bits.

More details can be found in the Proof of Concept (*exploit_4byte.cpp*) source code enclosed to this paper.

1-byte write-what-where exploitation

As it turns out, overwriting a 32-bit value is even more than is actually required to conduct a successful attack. As the previous section states, the number of bits we initially control is 52 – the remaining 12 bits are normally beyond our control (containing critical information like descriptor type, DPL Present flag etc). If we

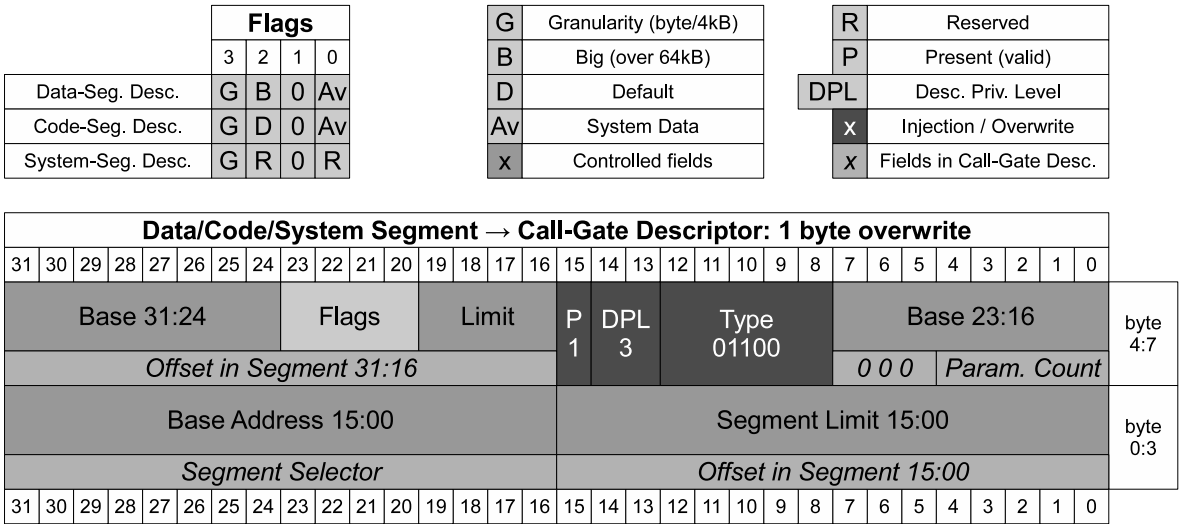

Figure 3: Data and Call-Gate Descriptors compared (one byte controlled)

manage to modify another 8 bits using the *write-what-where* vulnerability, only 4 uncontrolled bits remain!

Offset in Segment 31:16	**The upper WORD of the Call-Gate routine, i.e. the payload**
(P)resent flag	1
DPL	11
Descriptor Type	1100
Param Count	0 (irrelevant)

Table 2: The desired flag values in a Call-Gate descriptor

Due to the fact that it is impossible to transform a data descriptor into a call gate without modifying the Descriptor Type field itself, we have no other choice than to "use" the eight additional, controlled bits on changing the 6th structure byte. The 4 bits that are left uncontrolled belong to the "*Offset in Segment 31:16*" address field – these bits are bit 21, bit 22, bit 23 and bit 24 of the routine address. This means, more or less, that the code execution might be forwarded to a memory area across 15,728,640 bytes (15MB), and we do not know the exact address. Whoever has heard about the heap spraying technique might already know a solution to this problem.

It is possible to easily allocate a continuous memory area sized 15MB, by invoking either standard API or undocumented functions (i.e. *VirtualAlloc* or *NtAllocateVirtualMemory*). After we've got this memory set up, a well-known *nop slide* technique can be employed in order to "catch" the execution flow at whatever address it lands on, and direct the path to the final payload code, placed at the beginning of the 16th megabyte. In this particular case, however, we decided to use the *jmp trampoline* technique with 16 "jmp" instructions directly pointing to the payload – each placed at the beginning of a 1MB memory area – Figure 4 should give you a better understanding of the "jump table" memory layout.

For more details, please refer to the *exploit_1byte.cpp* source code, illustrating the process of converting a valid data descriptor into a Call-Gate.

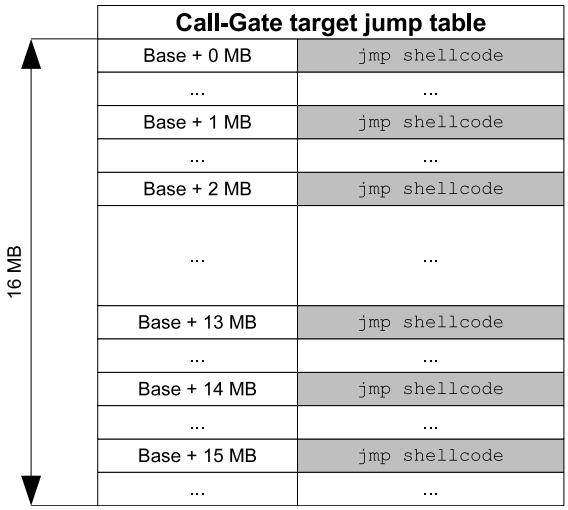

Figure 4: "Jump traps" amortizing the unknown address bits

Custom LDT goes User Mode

So far we have looked through some possible ways of playing with the LDT contents in order to escalate the execution privileges. However, as explained before, the *Local Descriptor Table* itself is a part of GDT, having

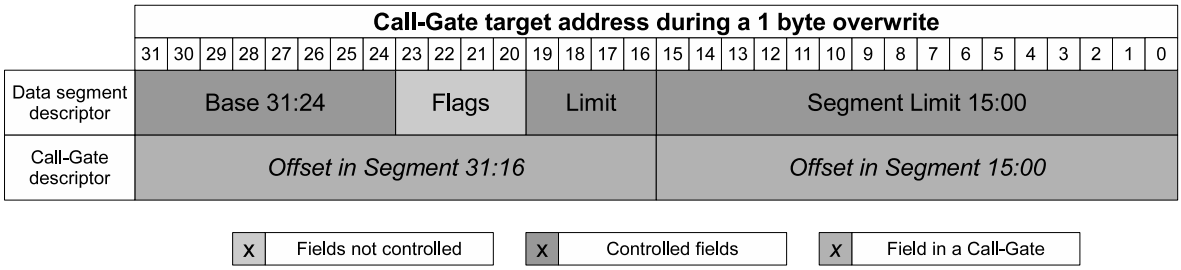

Figure 5: Call-Gate address assembled in Data Segment and Call-Gate Descriptors

its own selector (stored in the LDTR register), descriptor and other settings. Therefore, instead of modifying what is *inside* LDT, one could simply try to begin with redirecting the LDT address in the first place! By doing so, one could get a chance to make the LDT descriptor point to user mode memory. Because of the fact that this is a per-process structure, there is no chance for an unexpected BSoD in the mean time; the address defined by the attacker will only be used when his application desires to do so.

If we actually managed to forward LDT to a controlled memory area, we would get automatic control over the entire table contents. Therefore, any number of Call-Gates / other descriptors could be created and successfully used by our code (such as CALL FAR instructions). The only question here is – is it possible to perform the redirection by modifying, again, only 4 (out of 8) bytes of the LDT entry? The answer is – YES, it is possible indeed. To get a better view of what is the target now, the reader is advised to take a look at Figure 1, System Descriptor.

In general, it would be best to modify as few bytes as possible – the only field to alter here is the descriptor Base Address. However, since we're supposed to overwrite memory using 4 bytes (or it's multiplicity), there is no other option but to change the whole upper DWORD of the descriptor (including "Base 31:24", "Limit 19:16", "Base 23:16" and some flags). On the other hand, we are able to get the original LDT descriptor easily, then operate only on the fields that we are interested in (i.e. the upper 16 bits of Base Address).

However, some important facts about how Windows handles the GDT during context switches, must be taken into consideration. Since the LDT is a per-process mechanism, each thread should have access to the table owned by its process. On the other hand, there are only n GDTs (n = number of processors) present in the system. Therefore, during every single thread switch, some volatile descriptors (such as the FS: data segment or LDT) are replaced with the ones defined by the thread to be executed. As can be seen, some GDT entries are non-permanent and have to be stored somewhere else in memory (such as internal thread/process structures), and put into the global table when needed.

For example, if one decided to overwrite a volatile GDT entry, it would remain modified until a context switch would occur (which happens pretty often). In order to change such a descriptor in a permanent manner, it is required to find the exact localization, where the original value is stored, in the first place. In the case of the LDT, one can find the following field in the undocumented KPROCESS structure:

```
nt!_KPROCESS
    +0x000 Header              : _DISPATCHER_HEADER
    +0x010 ProfileListHead     : _LIST_ENTRY
    +0x018 DirectoryTableBase  : Uint4B
    +0x01c LdtDescriptor       : _KGDTENTRY
    +0x024 Int21Descriptor     : _KIDTENTRY
    +0x02c ThreadListHead      : _LIST_ENTRY
(...)
```

Therefore, we need to find the current process' KPROCESS structure and overwrite a part of the *LdtDescriptor* field. When it's done, a single context switch will do the rest of job; when our thread regains execution, the modified LDT descriptor will already be loaded. The question is, however, how to find the internal structure describing the current process; the solution is very simple: **NtQuerySystemInformation**(*SystemHandleInformation*).

Windows kernel makes it possible for every application to retrieve the complete list of all object handles present in the system (gathered from all processes), through the undocumented *NtQuerySystemInformation* routine together with the *SystemHandleInformation* request type. A structure describing a single HANDLE includes the creator process ID, handle value, and most importantly, the kernel object address. As the Process Objects are associated with the KPROCESS structure itself, the following steps make it pos-

sible for us to calculate the precise address of the *LdtDescriptor* field inside the current process' kernel-mode structure:

1. Open a real handle to the self-process by duplicating the *GetCurrentProcess*() pseudo-handle,
2. Call the NtQuerySystemInformation function,
3. Go through the returned list in search of the desired CreatorProcessId and Handle values,
4. Return the *Object* field plus *LdtDescriptor* offset (0x20 on Windows XP/Vista, 0x1C on Windows7).

Right after performing the overwrite, the application gains full control over the LDT contents; it can create Data / Code / Call-Gate / other types of descriptors without any restrictions. The specific way of how the elevation of privileges is achieved from this point on is up to the attacker; one possible way is to follow the exploitation steps covered in the previous section – setting up a valid Call-Gate in the context of user-mode memory should no longer be a problem.

Summary

Kernel mode vulnerabilities have been around for a significant amount of time; however, they're riveting more and more attention of the security world, recently. As the number of vulnerabilities found in device drivers and standard system modules is constantly growing, so are the exploitation techniques. Because of the fact that the majority of Windows users are spread across Windows XP, Windows Vista and Windows 7, exploitation compatibility might often pose a problem for exploit developers. In order to minimize the number of version-dependant mechanisms, it is recommended to take advantage of the hardware specifics rather than rely on undocumented system behavior that might turn out to be changed in the future or simply not work under certain circumstances.

By using the *Global* and *Local Descriptor Tables* as the *write-what-where* target, one can be sure that not only will his exploit stably work regardless of the platform being attacked, but also is very likely to be compatible with any future x86 Windows edition.

References

[1] GetThreadSelectorEntry Function. `http://msdn.microsoft.com/en-us/library/ms679363.aspx`.

[2] Intel Manuals. `http://www.intel.com/products/processor/manuals/`.

[3] Piotr Bania. Exploiting Windows Device Drivers. `http://pb.specialised.info/all/articles/ewdd.pdf`.

[4] Ruben Santamarta. Exploiting Common Flaws in Drivers. `http://reversemode.com/index.php?option=com_content&task=view&id=38&Itemid=1`.

[5] SLDT - Store Local Descriptor Table Register. `http://faydoc.tripod.com/cpu/sldt.htm`.

[6] SoBeIt. How to exploit Windows kernel memory pool. `http://packetstormsecurity.nl/Xcon2005/Xcon2005_SoBeIt.pdf`.

[7] Z0mbie. Adding LDT Entries In Win2K. `http://vx.netlux.org/lib/vzo13.html`.

Attachments

The Proof of Concept source code is attached to this PDF. Alternatively the source code can be downloaded from `http://vexillium.org/dl.php?ldtsource.zip`.