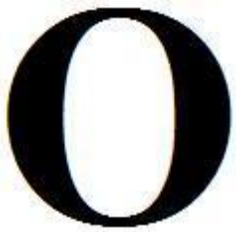
The oark book: http://code.google.com/p/oark/



The Open Source Anti Rootkit

David Reguera Garcia aka Dreg - Dreg@fr33project.org

Co-authors: -

January 31, 2011

Contents

Chapter 1
Call Gates and GDT/LDT

1.1 GDT/LDT

The Global Descriptor Table or GDT (extracted from Wikipedia[1]) is a data structure used by Intel x86-family processors in order to define the characteristics of the various memory areas used during program execution, for example the base address, the size and access privileges like executability and writability. These memory areas are called segments in Intel terminology.[2]

The GDT can hold things other than segment descriptors as well. Every 8-byte entry in the GDT is a descriptor, but these can be Task State Segment (or TSS) descriptors, Local Descriptor Table (LDT) descriptors, or Call Gate descriptors. The last one, Call Gates, are particularly important for transferring control between x86 privilege levels although this mechanism is not used on most modern operating systems.

There is also an LDT or Local Descriptor Table. The LDT is supposed to contain memory segments which are private to a specific program, while the GDT is supposed to contain global segments. The x86 processors contain facilities for automatically switching the current LDT on specific machine events, but no facilities for automatically switching the GDT.

Every memory access which a program can perform always goes through a segment. On the 386 processor and later, because of 32-bit segment offsets and limits, it is possible to make segments cover the entire addressable memory, which makes segment-relative addressing transparent to the user.

In order to reference a segment, a program must use its index inside the GDT or the LDT. Such an index is called a segment selector or selector in short. The selector must generally be loaded into a segment register to be used. Apart from the machine instructions which allow one to set/get the position of the GDT (and of the Interrupt Descriptor Table) in memory, every machine instruction referencing memory has an implicit Segment Register, occasionally two. Most of the time this Segment Register can be overridden by adding a Segment Prefix before the instruction.

Loading a selector into a segment register automatically reads the GDT or the LDT and stores the properties of the segment inside the processor itself. Subsequent modifications to the GDT or LDT will not be effective unless the segment register is reloaded.

1.2 Call Gates

A Call Gate [Inta] is a mechanism in the Intel x86 architecture to change privilege levels of the CPU when running a predefined function that is called by the instruction CALL/JMP FAR.

A call to a Call Gate allows you to obtain higher privileges than the current, for example we can execute a routine in ring0 using a CALL FAR in ring3. A Call Gate is an entry in

[1] Wikipedia. "Global Descriptor Table". In: (2010). URL: http://en.wikipedia.org/wiki/Global_Descriptor_Table.

[2] *Intel 64 and IA-32 Architectures, Software Developers Manual, System Programming Guide, Part 1.* URL: http://www.intel.com/design/processor/manuals/253668.pdf.

the GDT (Global Descriptor Table) or LDT (Local Descriptor Table). There are a GDT for each CORE, and each GDT can have one or more LDTs[3].

Windows doesn't use Call Gate for anything special, but there are malware, as the worm Gurong.A[4] , that installs a Call Gate via DevicePhysicalMemory to execute code on ring0. An article that talks about it is "Playing with Windows/dev/(k)mem"[5].

Nowadays we can't easily access to /Device/PhysicalMemory, I recommend reading the presentation by Alex Ionescu at RECON 2006 "Subverting Windows 2003 SP1 Kernel Integrity Protection"[6]. Also, there are examples[7] in the wired that use the API ZwSystemDebugControl[8] to install a Call Gate, but Ionescu's article says that it doesn't work nowadays (although there are techniques to reactivate them).

Gynvael and j00ru made a Call-Gate mechanism in kernel/driver exploit development on Windows[9], or, to be more precise, to use a write-what-where condition to convert a custom LDT entry into a Call-Gate (this can be done by modifying just one byte), and using the Call-Gate to elevate the code privilege from user-land to ring0.

David Reguera Garcia aka Dreg made a Call Gate detector for the free anti-rootkit Rootkit Unhooker[10]. The next releases have new features to detect other new stuff like the 'new LDT Forward to user mode attack' (published also in the Gynvael and j00ru's paper). With this attack, an attacker can add a Call Gate or other descriptor to LDT without restrictions from user mode.

An entry in the GDT/LDT looks like this:

```
typedef struct _SEG_DESCRIPTOR
{
    WORD size_00_15;
    WORD baseAddress_00_15;
    WORD baseAddress_16_23:8;
    WORD type:4;
    WORD sFlag:1;
    WORD dpl:2;
    WORD pFlag:1;
    WORD size_16_19:4;
    WORD notUsed:1;
    WORD lFlag:1;
```

[3]David Reguera Garcia aka Dreg. "Rootkit Arsenal, Installing a Call Gate (English translation of the Spanish post in blog.48bits.com)". In: (2010). URL: http://www.rootkit.com/blog.php?newsid=992.

[4]F-secure. "W32/Gurong.A: A worm in e-mails and in Kazaa shared folders. It has a rootkit functionality. This worm appeared on the 21st of March 2006." In: (2006). URL: http://www.f-secure.com/v-descs/gurong_a.shtml.

[5]crazylord. "Playing with Windows /dev/(k)mem". In: (2002). URL: http://www.phrack.com/issues.html?issue=59&id=16.

[6]Alex Ionescu. "RECON 2006: Subverting Windows 2003 SP1 Kernel Integrity Protection". In: (2006). URL: http://www.alex-ionescu.com/recon2k6.pdf.

[7]SACCOPHARYNX. "Call Gates". In: (2006). URL: http://ricardonarvaja.info/WEB/OTROS/TUTES%20SACCOPHARYNX/#Ring0.

[8]Undocumented NT Internals. *Function NtSystemDebugControl is used by some low-level debuggers written by Microsoft and avaiable typically in DDK*. URL: http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/Debug/NtSystemDebugControl.html.

[9]Matthew j00ru Jurczyk and Gynvael Coldwind. "GDT and LDT in Windows kernel vulnerability exploitation". In: (2010). URL: http://gynvael.coldwind.pl/?id=274.

[10]DiabloNova aka EP X0FF. "Rootkit Unhooker LE 3.8.386.588 SR1". In: (2010). URL: http://www.rootkit.com/blog.php?newsid=993.

```
    WORD DB:1;
    WORD gFlag:1;
    WORD baseAddress_24_31:8;
} SEG_DESCRIPTOR, *PSEG_DESCRIPTOR;
```

Listing 1.1: GDT/LDT Descriptor structure

A Call Gate is an entry type in the GDT/LDT which has the following appearance:

```
typedef struct _CALL_GATE_DESCRIPTOR
{
    WORD offset_00_15;
    WORD selector;
    WORD argCount:5;
    WORD zeroes:3;
    WORD type:4;
    WORD sFlag:1;
    WORD dpl:2;
    WORD pFlag:1;
    WORD offset_16_31;
} CALL_GATE_DESCRIPTOR, *PCALL_GATE_DESCRIPTOR;
```

Listing 1.2: Call Gate Descriptor structure

- **offset_00_15:** is the bottom of the address of the routine to be executed in ring0, **offset_16_31** is the top.
- **selector:** specifies the code segment with the value KGDT_R0_CODE (0x8), the routine will run ring0 privileges.
- **argCount:** the number of arguments of the routine in DWORDs.
- **type:** the descriptor type for a 32-bit Call Gate needs the value 0xC
- **dpl:** minimum privileges that the code must have to call the routine, in this case 0x3, because it will be called by the routine ring3

To create a Call Gate we can follow the following steps:

1. Build the Call Gate that points to our routine.
2. Set the code only in a core (remember: there are a GDT for each CORE.)
3. Read the GDTR register in order to find the GDT address and the size using SGDT instruction:

```
typedef struct _GDTR
{
    WORD nBytes;
    DWORD baseAddress;
} GDTR;
```

Listing 1.3: GDTR register

We can obtain the number of entries (number of GDT descriptors) with GDTR.nBytes/8.

4. Find a free entry in the GDT/LDT.
5. Write the Call Gate descriptor.
6. To call the Call Gate is only necessary to make a CALL/JMP FAR to the GDT/LDT selector:
   - ie if we've introduced the Call Gate at the entry **100** of the **GDT**, the user space application must execute a CALL/JMP FAR **0x320**:00000000. 0x320 is in binary 1100100 **0** 00, then the entry is:1100100 (100 in binary is 1100100), **TI=0** (entry is in **GDT**) RPL=00.. The other part of the FAR CALL is not useful but must be in the instruction.
   - ie if we've introduced the Call Gate at the entry **100** of the **LDT**, the user space application must execute a CALL/JMP FAR **0x324**:00000000. 0x324 is in binary 1100100 **1** 00, then the entry is:1100100 (100 in binary is 1100100), **TI=1** (entry is in **LDT**) RPL=00..

1.2.1 Windows specific

Now is time for a more detailed view. On Windows is easy to control in which core do you want run the code, it is only necessary two APIs, to get the number of COREs you can use GetSystemInfo:

```
void WINAPI GetSystemInfo( LPSYSTEM_INFO lpSystemInfo );
```

Listing 1.4: API to get the system info

SYSTEM_INFO has the following appearance:

```
typedef struct _SYSTEM_INFO {
    // .............
    DWORD dwNumberOfProcessors;
    // .............
}SYSTEM_INFO;
```

Listing 1.5: SYSTEM INFO structure

We do a loop with the field dwNumberOfProcessors that executes CORE by CORE by adding the Call Gate or you can also force the Call Gate only to the first core (1) and the user space application will be executed only in the core 1, this is accomplished using the API: SetThreadAffinityMask, as follows:

```
DWORD_PTR WINAPI SetThreadAffinityMask( HANDLE hThread, DWORD_PTR
    dwThreadAffinityMask );
```

Listing 1.6: SetThreadAffinityMask API

Passing a GetCurrentThread() and the value 1 as AffinityMask, Be careful, DWORD_PTR is not a pointer to a DWORD, is passed by value:

```
Affinity = 1;
SetThreadAffinityMask( GetCurrentThread(), Affinity );
```

Listing 1.7: SetThreadAffinityMask to core 1

You can do a loop for the number of processors and the rate variable (the first core is 1 not 0) using Affinity you can install a Call Gate in all the cores. Take notice that it's a mask, so you should use a bit shift like: AffinityMask = 1 ¡¡ index_variable to scroll through the cores.

To install a Call Gate from a driver your need other APIs, to obtain the number of cores:

```
ZwQuerySystemInformation( SystemBasicInformation, &
    system_basic_information, sizeof( system_basic_information ), NULL );
```

Listing 1.8: Get the number of cores from a driver

To change the core, -2 is the equivalent to GetCurrentThread():

```
ZwSetInformationThread( (HANDLE) -2, ThreadAffinityMask, & AffinityMask,
    sizeof( AffinityMask ) );
```

Listing 1.9: Change the affinity from a driver

With this information you can install/detect call gates in the GDTs and LDTs of all cores. But for the best detection it is necessary view all LDTs, and the Windows Scheduler uses the same entry in the GDT for all processes, and only with the last information you can loss some LDTs.

LDT specific

Windows Scheduler uses the same entry in the GDT for LDT in all processes and it is necessary read the LDTs of all processes, there are two ways: Using the Windows API and DKOM (Direct Kernel Object Manipulation).

Chapter 2
PEB Hooking

PEB HOOKING method consists in supplanting a DLL in memory using a fake DLL, so all modules of a process now will use the fake DLL. Deroko writes in ARTeam 2 "PEB DLL Hooking Novel method to Hook DLLs"[1] the first public article about PEB Hooking (including a POC). After, Juan Carlos Montes and David Reguera Garcia writes an article for Phrack 65 called: phook - The PEB Hooker[2], this article have more information and other tools. After David Reguera Garcia writes other article[3] and a new engine called dwtf[4] to generate Fake DLLs at runtime, in the past you needed a fake DLL repository for the different DLLs versions, a useful engine for the system DLLs (Windows SP1 have different system DLLs than Windows SP2 etc).

Process Environment Block (PEB) is a structure located in the user's space, that contains the process' enviroment data: Enviroment variables. Loaded modules list...

```
typedef struct _PEB
{
    //...
    PVOID ImageBaseAddress;
    PPEB_LDR_DATA LoaderData;
    //...
} PEB;
```

Listing 2.1: Process Environment Block

For PEB HOOKING we need use the LoaderData field, this field is a structure in which there are some data about the modules of a process. It is a doubly linked list and it can be sorted by Order of: loading, in memory and initialization. All flink and blink fields in LIST_ENTRY are in reality pointers to LDR_MODULE. We are going to manipulate from LDR_MODULE: BaseAddress, EntryPoint and SizeOfImage.

It is necessary to search DLL_FAKE and DLL_REAL for some identificative fields of LDR_MODULE, once found the following data will be exchanged: EntryPoint, BaseAddress and SizeOfImage.

```
typedef struct _PEB_LDR_DATA
{
    ...
    //BOOLEAN Initialized;
    PVOID SsHandle;
    LIST_ENTRY InLoadOrderModuleList;
    //....
}
```

Listing 2.2: PEB Loader Data

[1] Deroko. "PEB Dll Hooking Novel method to Hook Dlls". In: *ARTeam* 2 (2006). URL: http://arteam.accessroot.com/arteam/site/download.php?view.275.

[2] David Reguera Garcia aka Dreg and Juan Carlos Montes Senra aka Shearer. "phook - The PEB Hooker". In: *Phrack* 65 (2008). URL: http://www.phrack.org/issues.html?issue=65&id=10.

[3] David Reguera Garcia aka Dreg. "Generating any DLL for PEB Hooking or replacing in disk, binary form". In: (2009). URL: http://www.rootkit.com/blog.php?newsid=988.

[4] David Reguera Garcia aka Dreg. "dwtf engine: creating a fake DLL at runtime." In: (2010). URL: http://rootkitanalytics.com/tools/dwtf.php.

```
typedef struct _LDR_MODULE
{
    LIST_ENTRY InLoadOrderModuleList;
    // ...
    PVOID BaseAddress;
    PVOID EntryPoint;
    ULONG SizeOfImage;
    // ...
} LDR_MODULE, *PLDR_MODULE;
```

Listing 2.3: Loader Module Data

Bibliography

[AK04] Chris Anley and Jack Koziol. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley, 2004. ISBN: 0764544683. URL: http://www.amazon.com/Shellcoders-Handbook-Discovering-Exploiting-Security/dp/0764544683.

[Bak00] Art Baker. *The Windows 2000 Device Driver Book: A Guide for Programmers (2nd Edition)*. Prentice Hall, 2000. ISBN: 0130204315. URL: http://www.amazon.com/Windows-2000-Device-Driver-Book/dp/0130204315.

[Blu09] Bill Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones And Bartlett Publishers, 2009. ISBN: 1598220616. URL: http://www.amazon.com/Rootkit-Arsenal-Escape-Evasion-Corners/dp/1598220616.

[cra02] crazylord. "Playing with Windows /dev/(k)mem". In: (2002). URL: http://www.phrack.com/issues.html?issue=59&id=16.

[Dab99] Prasad Dabak. *Undocumented Windows NT*. Hungry Minds, 1999. ISBN: 0764545698. URL: http://www.amazon.com/Undocumented-Windows-NT%C2%AE-Prasad-Dabak/dp/0764545698.

[Der06] Deroko. "PEB Dll Hooking Novel method to Hook Dlls". In: *ARTeam* 2 (2006). URL: http://arteam.accessroot.com/arteam/site/download.php?view.275.

[Dre09] David Reguera Garcia aka Dreg. "Generating any DLL for PEB Hooking or replacing in disk, binary form". In: (2009). URL: http://www.rootkit.com/blog.php?newsid=988.

[Dre10a] David Reguera Garcia aka Dreg. "dwtf engine: creating a fake DLL at runtime." In: (2010). URL: http://rootkitanalytics.com/tools/dwtf.php.

[Dre10b] David Reguera Garcia aka Dreg. "Rootkit Arsenal, Installing a Call Gate (English translation of the Spanish post in blog.48bits.com)". In: (2010). URL: http://www.rootkit.com/blog.php?newsid=992.

[DS08] David Reguera Garcia aka Dreg and Juan Carlos Montes Senra aka Shearer. "phook - The PEB Hooker". In: *Phrack* 65 (2008). URL: http://www.phrack.org/issues.html?issue=65&id=10.

[EC05] Eldad Eilam and Elliot J. Chikofsky. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005. ISBN: 0764574817. URL: http://www.amazon.com/Reversing-Secrets-Engineering-Eldad-Eilam/dp/0764574817/ref=pd_sim_b_2.

[Eri08] Jon Erickson. *Hacking: The Art of Exploitation, 2nd Edition*. No Starch Press, 2008. ISBN: 1593271441. URL: http://www.amazon.com/Hacking-Art-Exploitation-Jon-Erickson/dp/1593271441/ref=pd_sim_b_12.

[EX10] DiabloNova aka EP X0FF. "Rootkit Unhooker LE 3.8.386.588 SR1". In: (2010). URL: http://www.rootkit.com/blog.php?newsid=993.

[Fs06] F-secure. "W32/Gurong.A: A worm in e-mails and in Kazaa shared folders. It has a rootkit functionality. This worm appeared on the 21st of March 2006." In: (2006). URL: http://www.f-secure.com/v-descs/gurong_a.shtml.

[Har10] Johnson M. Hart. *Windows System Programming (4th Edition)*. Addison-Wesley Professional, 2010. ISBN: 0321657748. URL: http://www.amazon.com/Windows-Programming-Addison-Wesley-Microsoft-Technology/dp/0321657748/ref=pd_sim_b_19.

[HB05] Greg Hoglund and James Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005. ISBN: 0321294319. URL: http://www.amazon.com/Rootkits-Subverting-Windows-Greg-Hoglund/dp/0321294319.

[Inta] *Intel 64 and IA-32 Architectures, Software Developers Manual, System Programming Guide, Part 1*. URL: http://www.intel.com/design/processor/manuals/253668.pdf.

[Intb] Undocumented NT Internals. *Function NtSystemDebugControl is used by some low-level debuggers written by Microsoft and avaiable typically in DDK*. URL: http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/Debug/NtSystemDebugControl.html.

[Ion06] Alex Ionescu. "RECON 2006: Subverting Windows 2003 SP1 Kernel Integrity Protection". In: (2006). URL: http://www.alex-ionescu.com/recon2k6.pdf.

[JC10] Matthew j00ru Jurczyk and Gynvael Coldwind. "GDT and LDT in Windows kernel vulnerability exploitation". In: (2010). URL: http://gynvael.coldwind.pl/?id=274.

[Nag97] Rajeev Nagar. *Windows NT File System Internals : A Developer's Guide*. O'Reilly Media, 1997. ISBN: 1565922492. URL: http://www.amazon.com/Windows-File-System-Internals-Developers/dp/1565922492.

[One02] Walter Oney. *Programming the Microsoft Windows Driver Model*. Microsoft Press, 2002. ISBN: 0735618038. URL: http://www.amazon.com/Programming-Microsoft-Windows-Driver-Model/dp/0735618038.

[Orw07] Penny Orwick. *Developing Drivers with the Windows Driver Foundation*. Microsoft Press, 2007. ISBN: 0735623740. URL: http://www.amazon.com/Developing-Drivers-Windows-Foundation-Developer/dp/0735623740.

[PH07] Daniel Pravat and Mario Hewardt. *Advanced Windows Debugging*. Addison-Wesley Professional, 2007. ISBN: 0321374460. URL: http://www.amazon.com/Advanced-Windows-Debugging-Mario-Hewardt/dp/0321374460/ref=pd_sim_b_24.

[Ree10] Ron Reeves. *Windows 7 Device Driver*. Addison-Wesley Professional, 2010. ISBN: 0321670213. URL: http://www.amazon.com/Windows-Device-Addison-Wesley-Microsoft-Technology/dp/0321670213.

[RSI09] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition (PRO-Developer)*. Microsoft Press; 5th ed. edition, 2009. ISBN: 0735625301. URL: http://www.amazon.com/Windows%C2%AE-Internals-Including-Windows-PRO-Developer/dp/0735625301/ref=pd_sim_b_3.

[SAC06] SACCOPHARYNX. "Call Gates". In: (2006). URL: http://ricardonarvaja.info/WEB/OTROS/TUTES%20SACCOPHARYNX/#Ring0.

[Sch01] Sven B. Schreiber. *Undocumented Windows 2000 Secrets: A Programmer's Cookbook*. Addison-Wesley Professional, 2001. ISBN: 0201721872. URL: http://www.amazon.com/Undocumented-Windows-2000-Secrets-Programmers/dp/02017218722.

[Szo05] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005. ISBN: 0321304543. URL: http://www.amazon.com/Art-Computer-Virus-Research-Defense/dp/0321304543/ref=sr_1_1?ie=UTF8&s=books&qid=1296406709&sr=8-1.

[Vie07] Ric Vieler. *Professional Rootkits (Programmer to Programmer)*. Wrox, 2007. ISBN: 0470101547. URL: http://www.amazon.com/Professional-Rootkits-Programmer-Ric-Vieler/dp/0470101547.

[Wik10] Wikipedia. "Global Descriptor Table". In: (2010). URL: http://en.wikipedia.org/wiki/Global_Descriptor_Table.