Are the conclusions drawn from observation of `Redpill` results wrong?

- Background on `SIDT`
- Wrong assumptions, wrong conclusions?
- My conclusions
- What `SIDTcon` does ...

Created: 2007-04-01

Author: Oliver Schneider (assarbad.net)

Preface

Almost everyone in the security community is aware of Joanna Rutkowska's tool `Redpill`. The article[1] is available on her website at `http://invisiblethings.org/papers/redpill.html`.

The method was quite impressive and I remember when I heard of it the first time. What is important about this method is the fact that the `SIDT` instruction is not privileged and can therefore be called from user mode[2].
Now Joanna Rutkowska claimed in her article:

> Because there is only one IDTR register, but there are at least two OS running concurrently (i.e. the host and the guest OS), VMM needs to relocate the guest's IDTR in a safe place, so that it will not conflict with a host's one. Unfortunately, VMM cannot know if (and when) the process running in guest OS executes SIDT instruction, since it is not privileged (and it doesn't generate exception). Thus the process gets the relocated address of IDT table. It was observed that on VMWare, the relocated address of IDT is at address `0xff`*`XXXXXX`*, whereas on Virtual PC it is `0xe8`*`XXXXXX`*. This was tested on VMWare Workstation 4 and Virtual PC 2004, both running on Windows XP host OS.

Throughout this short paper I am attempting to prove that this is a wrong *conclusion*. My tests have been run with the currently[3] latest version of VMWare Workstation.

Although Joanna Rutkowska got the most attention, others have previously used the descriptor tables such as GDT and IDT to detect the presence of a virtual machine monitor (VMM) and elaborated on the topic.

About the author

I am of German origin and currently live in Reykjavik (Iceland), where I work for FRISK Software International, creators of *F-Prot Antivirus*, as researcher and developer.

In my spare time I enjoy programming, reverse engineering, reading books, learning foreign languages and drinking a good brandy or cognac.

[1] ... as well as the tool's source code
[2] CPL3 or ring 3 as it is called also in some documents.
[3] ... as of 2007-04-01

Chapter 1

Background on SIDT

Calling SIDT is an interesting thing. First of all you have to be aware of the fact that the interrupt descriptor table, or IDT, exists for *each* processor [1]. This also means you need some way to determine the address to the IDT of all the processors in your system.

Not only that. Since a "normal" user mode process is not usually bound to one of the processors, you have to have a way to force it to run on a certain processor and then retrieve the address to the IDT. On Windows, our "specimen", this can be done via the SetProcessAffinityMask() API function.

For you to understand what we are going to execute, here is the code to read the address of the IDT and return it to the caller. There we go:

```
ULONG_PTR GetIdtBaseAddress()
{
    #pragma pack(1)
    struct { USHORT Limit; ULONG_PTR BaseAddress; } idtr;
    #pragma pack()
    _asm sidt idtr;
    return idtr.BaseAddress;
}
```

Very compact and not too hard to understand, I think. We define a structure which resembles the IDTR structure and ask via the assembly instruction SIDT to store the contents of the IDTR into our structure and then return the address part of it to our caller.

Joanna Rutkowska was using a slightly different approach, stuffing everything together into one function and trying to avoid assembly code parts for "portability". Her code looks like this:

```
int swallow_redpill ()
{
   unsigned char m[2+4], rpill[] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
   *((unsigned*)&rpill[3]) = (unsigned)m;
   ((void(*)())&rpill)();
   return (m[5]>0xd0) ? 1 : 0;
}
```

Now, her code does already a little bit more which could be described by the following function that calls GetIdtBaseAddress():

[1] ... or core for that matter.

```
int swallow_redpill ()
{
    return (GetIdtBaseAddress() > 0xD0000000);
}
```

So if the IDT base address is at a higher position than `0xD0000000`, she concludes to be inside a virtual machine. This conclusion is wrong, even if we would assume for a moment that her claim about the relocated GDTR/IDTR - see the foreword - are right. But despite that, the problem would already arise on multi-processor machines where a test run could give reasonably high addresses for one processor and "normal" ones for the other. Since the `redpill.c` does not take this into account the result is *per-se* unreliable. I have also seen some papers that attempted to call the instruction a number of times under the assumption that the result would show an even distribution between the processors. In my opinion this is also not quite the best approach, given that one can easily set the affinity of the process without special privilege requirements.

1.1 Why does setting the affinity work?

Setting the affinity works for the current process and all its threads. Now why would a driver stick to this setting anyway? For two reasons:

- I have not introduced any code into the driver which would change this setting.
- This driver sits most likely on top of a driver stack - except someone attached a filter driver to it[2]. Also since we use `DeviceIoControl()` to talk to the driver we can be certain that the thread context remains the same.
  One misconception of many people is, that a driver is something like a "program" in user mode. Not so, the driver is more like a DLL and gets called by different threads inside different processes.
  `DeviceIoControl()` is a direct channel to the driver and therefore we can guarantee that the thread context is stable even though we switch CPL[3].

[2] Highly unlikely that someone attaches a filter during the short time the driver is loaded. It gets immediately unloaded after it has done its job.

[3] Meaning we are switching to CPL0, or ring 0 as it is sometimes called: in short, kernel mode.

Chapter 2

Wrong assumptions, wrong conclusions?

2.1 "VMM needs to relocate the guest's IDTR in a safe place"

If this was the case, one could easily prove it, right? Test it yourself by running multiple virtual machines concurrently and read out the values of the IDT address using my tool `SIDTcon`[1].

If the IDTR is relocated, why can both virtual machines[2] in such a test have the same value for the IDT address? But it gets better.

2.2 "Unfortunately, VMM cannot know if (and when) the process running in guest OS executes SIDT instruction"

This quote implies that VMWare passes all non-privileged instructions on to the host and only catches privileged ones because they cause a trap. This can be falsified easily - but how?

2.3 Running `SIDTcon`, getting strange results ..

If you have run my program `SIDTcon` as I asked you to, you will likely have noticed strange values and strange abbreviations. Here is a sample output from the machine[3] where I am typing this text.

```
SIDTcon - demonstration of SIDT discrepancies
(c) 2006-2007 by Oliver Schneider (assarbad.net)

Operating System Version 5.1.2600 (probably not in a VMM)
Multi-processor system recognized, will retrieve info per-processor!

Processor #00:
--------------
        (UM)IDT base address: 8003F400 (2047)
```

[1] A detailed description of the tool follows.
[2] Given it is the same operating system inside the virtual machine.
[3] ... which is not a virtual machine.

```
        (UM)GDT base address: 8003F000 (1023)
        (KM)IDT base address: 8003F400 (2047)
        (KM)GDT base address: 8003F000 (1023)

Processor #01:
--------------
        (UM)IDT base address: F772A560 (2047)
        (UM)GDT base address: F772A160 (1023)
        (KM)IDT base address: F772A560 (2047)
        (KM)GDT base address: F772A160 (1023)
```

From this output you can see the operating system version, the address of the IDT on both processors and the address of the global descriptor table (GDT) on both processors. The values look consistent. KM is used as the abbreviation for *kernel mode* and UM for *user mode*. The numbers in brackets are the table limits of IDT and GDT respectively.

If you wonder what the "`(probably not in a VMM)`" is about, it just uses Joanna Rutkowska's method to make a guess whether this is run inside a virtual machine (i.e. calling `swallow_redpill()`). Nothing arcane ...
Let us run the same program in a virtual machine now:

```
SIDTcon - demonstration of SIDT discrepancies
(c) 2006-2007 by Oliver Schneider (assarbad.net)

Operating System Version 5.1.2600 (probably inside a VMM)
Single-processor system recognized.

        (UM)IDT base address: FFC18000 (2047)
        (UM)GDT base address: FFC07000 (16687)
        (KM)IDT base address: 8003F400 (2047)
        (KM)GDT base address: 8003F000 (1023)
```

As we can see the operating system versions are the same[4] and the VMM only provides one processor to the guest OS. But what is that? The values for the addresses differ between user mode and kernel mode?! Even worse (for Joanna), the values of the IDT address of the first processor on the host is identical to that inside the virtual machine when read from kernel mode.

Now of course I hear the first people mumbling "Maybe the kernel mode result shows the host's IDT address?!". Nope. And why this is not the case can be easily seen from the following table:

[4] They are both XP, but the host is German and the guest English and while the host runs XP SP2, the guest runs XP without SP. This does not affect the IDT default address, though.

VMM	Windows	IDT (KM)	IDT (UM)	GDT (KM)	GDT (UM)
VMM tools installed - acceleration enabled					
VMWare	5.0.2195	80036400	FFC18000	80036000	FFC07000
	5.1.2600	8003F400	FFC18000	8003F000	FFC07000
	5.2.3790	8003F400	FFC18000	8003F000	FFC07000
Virtual PC	5.0.2195	80036400		80036000	
	5.1.2600	8003F400		8003F000	
	5.2.3790	8003F400		8003F000	
VMM tools installed - acceleration disabled					
VMWare	5.0.2195	80036400		80036000	
	5.1.2600	8003F400		8003F000	
	5.2.3790	8003F400		8003F000	
Virtual PC	5.0.2195	BDF98500		BDF98D00	
	5.1.2600	F9F98500		F9F98D00	
	5.2.3790	F8B98500		F8B98D00	
*VMM tools **not** installed - acceleration enabled*					
VMWare	5.0.2195	80036400	FFC18000	80036000	FFC07000
	5.1.2600	8003F400	FFC18000	8003F000	FFC07000
	5.2.3790	8003F400	FFC18000	8003F000	FFC07000
Virtual PC	5.0.2195	80036400		80036000	
	5.1.2600	8003F400		8003F000	
	5.2.3790	8003F400		8003F000	
*VMM tools **not** installed - acceleration disabled*					
VMWare	5.0.2195	80036400		80036000	
	5.1.2600	8003F400		8003F000	
	5.2.3790	8003F400		8003F000	
Virtual PC	5.0.2195	80036400	E8398500	80036000	E8398D00
	5.1.2600	8003F400	E5798500	8003F000	E5798D00
	5.2.3790	8003F400	E8398500	8003F000	E8398D00

Several notes are necessary. The VMWare Workstation version used was *5.5.3 Build 34685*, Virtual PC 2007 was *6.0.156.0*. All of the virtual machines have been tested one after another, so for this test I did not run two of them simultaneously! For Virtual PC 2007 the hardware virtualization features of the CPU on the host machine where used. The following table shows the respective values for the host system:

CPU	IDT (KM)	IDT (UM)	GDT (KM)	GDT (UM)
Multi-processor system running Windows XP SP2 (5.1.2600)				
#0	8003F400	8003F400	8003F000	8003F000
#1	F772A560	F772A560	F772A160	F772A160

It is obvious that the results between kernel mode and user mode are *always* consistent on the host machine, but *sometimes* show a discrepancy when retrieved from inside the virtual machine. But from this table one can also see another interesting fact. The address retrieved in kernel mode is **not** identical to the one of the host system (e.g. for the Windows 2000 guest).

Moreover a pattern seems to exist. Whereever the hardware virtualization features were used, the descriptor table address is the same as it would be on a physical machine. This is only valid for Virtual PC 2007, though, since VMWare does not provide support for hardware virtualizations in the 32bit version of their product and hence wasn't tested. For VMWare the native values were

observed in all cases where the acceleration was turned off. For Virtual PC 2007 the results where particularly interesting with the "Virtual Machine Additions" installed and disabled hardware virtualization. The results suggest that the "Virtual Machine Additions" have an influence on the addresses for the descriptor tables used by the operating system.

Please find the old test results in appendix A!

2.4 My conclusions

The conclusions by Joanna Rutkowska seem to be wrong given my research. However, there are some uncertainties here as well. First of all she had used an *older version* of VMWare and implementation details relevant for this research *may have changed over time*. Another point is that the research was mostly limited to VMWare and Virtual PC, which are only two out of the three most popular type II VMM vendors: VMWare, Microsoft, Parallels. So Parallels should be tested as well. What can be definitely stated, though, is that `Redpill` is far from reliable which can be attributed to different facts:

- It does not take into account multi-processor machines
- It works only in user mode for VMWare with enabled acceleration[5]
- The criteria need to be revised

2.5 Your help is needed

If you own an older version of VMware or a current one that has not been tested (e.g. with Linux as host OS), please run my `SIDTcon` tool inside a guest Windows and send me the results along with the version of VMWare, the host system version and the guest system version[6]. Also if you have other type II VMMs running, don't hesitate to contact me via my website. Thank you.

[5] It appears that Joanna Rutkowska is well aware of this, since her *System Virginity Verifier* (SVV) uses a kernel mode driver to retrieve the IDT address.

[6] I cannot guarantee that `SIDTcon` will work on Windows NT4, but it is supposed to run on 32bit Windows Vista.

Chapter 3

What SIDTcon does ...

SIDTcon is a very simple console based tool that relies on a simple legacy type NT driver to retrieve the kernel mode results for SIDT. The idea of it is simple. Just call SIDT and - out of curiosity - SGDT from kernel and from user mode, format the results and output them to the user.

The driver has been written in C++, because it provides stricter type checking. There is no other reason. I do not use any classes or so. The project was created with my free project creation wizard DDKWizard, which can be found on http://ddkwizard.assarbad.net.

3.1 Building SIDTcon and SIDTdrv

Download it first: http://assarbad.net/stuff/!export/SIDT.rar
To build both projects just open the Visual Studio 2005 solution and build it. Alternatively you can compile the projects directly with the DDK BUILD command or via one of the DDKBUILD scripts[1] from OSR. The scripts as well as the manual to DDKWizard which describes configuration of DDKBUILD and DDKWizard can be found on the DDKWizard-website.

If you use the Visual Studio method you will still need the Windows XP, Windows 2003 or Windows Vista DDK (or any variant of them). This is required along with one of the DDKBUILD scripts to build both, the driver and the console program. Yes, the console program is actually a DDK project! The source of the console application is heavily commented, so I will not discuss this part.

I will just drop a few words about the driver source. Both projects share the files in the .\common folder. The .h file has some declarations which are shared between user and kernel mode and the .cpp files contains two functions to fetch the addresses of the IDT and the GDT respectively.

Main entry point to the driver is the DriverEntry function which creates a device object and a symlink inside an object directory that is accessible from user mode. IRP_MJ_DEVICE_CONTROL and the respective dispatch function SIDTDRV_DispatchDeviceControl are doing the main job. SIDTDRV_DriverUnload's sole job is to ensure that the driver can be unloaded after it has done its work. SIDTDRV_DispatchCreateClose is a dummy dispatch function that becomes only interesting if the driver is being improved and handles more resources.

The dispatch routine which handles the IOCTLs from the user mode console application is as simple as can be:

[1] Actually this project might *require* the .cmd version of the script. Not tested, though.

```
NTSTATUS SIDTDRV_DispatchDeviceControl(
    IN PDEVICE_OBJECT       DeviceObject,
    IN PIRP                 Irp
    )
{
    PIO_STACK_LOCATION irpSp = IoGetCurrentIrpStackLocation(Irp);

    switch(irpSp->Parameters.DeviceIoControl.IoControlCode)
    {
    case IOCTL_GETBASEADDR:
        if(sizeof(BASE_ADDRESSES) == irpSp->Parameters.DeviceIoControl.OutputBufferLength)
        {
            PBASE_ADDRESSES lpAddr = PBASE_ADDRESSES(Irp->AssociatedIrp.SystemBuffer);
            lpAddr->IdtBaseAddress = GetIdtBaseAddress();
            lpAddr->GdtBaseAddress = GetGdtBaseAddress();
            lpAddr->uIdtLimit = GetIdtLimit();
            lpAddr->uGdtLimit = GetGdtLimit();
            Irp->IoStatus.Status = STATUS_SUCCESS;
            Irp->IoStatus.Information = sizeof(BASE_ADDRESSES);
        }
        break;
    default:
        Irp->IoStatus.Status = STATUS_INVALID_DEVICE_REQUEST;
        Irp->IoStatus.Information = 0;
        break;
    }

    NTSTATUS status = Irp->IoStatus.Status;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}
```

The most important part is the one, where the driver fetches the base addresses of the two descriptor tables and puts them into the user buffer:

```
PBASE_ADDRESSES lpAddr = PBASE_ADDRESSES(Irp->AssociatedIrp.SystemBuffer);
lpAddr->IdtBaseAddress = GetIdtBaseAddress();
lpAddr->GdtBaseAddress = GetGdtBaseAddress();
lpAddr->uIdtLimit = GetIdtLimit();
lpAddr->uGdtLimit = GetGdtLimit();
```

Not forgetting to set `Irp->IoStatus.Information` is self-explanatory and then the information goes already up to user mode where `SIDTcon` takes care of displaying it.

3.2 Running `SIDTcon`

`SIDTcon` requires `SIDTdrv.sys` to reside in the same directory as itself. This directory must not be a network share, since the driver is loaded from this location. Starting `SIDTcon` is straightforward. You don't need any command line parameters. Just run it and watch the output.

3.3 Troubleshooting

If SIDTcon is run from a network share the loading of the driver will likely fail. The same may hold for readonly drives (not tested).

3.4 License

The whole code is released into the public domain. This also means that I can not be held liable for *any* damage resulting from the use of SIDTcon or its components.

This, however, means that you can freely use any of the components of the program and the driver for any purpose including the use in commercial programs.

Chapter 4

Appendix A

Following are the old test results from October 2006 when revision 1 of this paper was released:

VMM	host	guest	IDT (KM)	IDT (UM)	GDT (KM)	GDT (UM)
Uni-processor host system running Windows XP SP2						
host (UP)	5.1.2600	-	8003F400	8003F400	8003F000	8003F000
VMW Wks 5	5.1.2600	5.0.2195	80036400	FFC18000	80036000	FFC07000
VMW Wks 5	5.1.2600	5.1.2600	8003F400	FFC18000	8003F000	FFC07000
VMW Wks 5	5.1.2600	5.2.3790	8003F400	FFC18000	8003F000	FFC07000
Multi-processor host system running Windows XP SP2						
host (MP)	5.1.2600	(CPU#0)	8003F400	8003F400	8003F000	8003F000
		(CPU#1)	F7732560	F7732560	F7732160	F7732160
VMW Srv 1	5.1.2600	5.0.2195	80036400	FFC18000	80036000	FFC07000
VMW Srv 1	5.1.2600	5.1.2600	8003F400	FFC18000	8003F000	FFC07000
VMW Srv 1	5.1.2600	5.2.3790	8003F400	FFC18000	8003F000	FFC07000

Several notes are necessary. The VMWare Workstation version used was *5.5.1 Build 19175* and VMWare Server was of version *1.0.1 Build 29996*. All of the virtual machines have been tested one after another, so for this test I did not run two of them simultaneously!

Contents