ATTENTION: Please note that this article is designed to be read parallel with the source code. Also I highly recommend to read the first uninformed article (link follows on next page), otherwise you won't understand much of the work covered here. This is no tutorial and beginners should not read it. Don't expect to understand all the stuff without working with the source code and the uninformed articles in parallel. There are so much things to know about PatchGuard that it is impossible to cover them all in one single article. Also, I don't want to repeat other guys work!

1 Introduction to PatchGuard

Before I start to explain how you can disable PatchGuard, I want to talk about some philosophic aspects.

If you are asking why one would disable PatchGuard, I could give you the following answers:

- It is challenging and to see how your invested time resulted in a driver, that you wrote yourself, and can disable a technology that should protect the kernel of a billion dollar OS and a trillion dollar company, really rewards all the effort. BTW you learn so much about the windows internals.
- Even if PatchGuard is a good thing for stability and security, it is still a two sided sword. PatchGuard is currently implemented in software only. Everyone will hopefully agree that this can't be secure. Well "secure", in what way? Of course Microsoft could defend all malware with plain software implementation and they already know how to do this when considering the Singularity project they are working on. What I mean by "PatchGuard is not secure" is that the end-user could disable it. This does not necessarily mean that malware can disable it. If PatchGuard remains as software implementation, there is nothing to worry about.
  But this is just the beginning. Look at Intel's LaGrande technology, the Next-Generation-Secure-Computing-Base (NGSCB, alias "Palladium"), the Protected-Media-Path and, well, the TPM as a passive component. The route is set clearly: Windows Vista 64-Bit is the most secure OS available for the usual guy and I hope many of you agree that we wouldn't need any further security, because we will pay for it with freedom; the freedom to use the computer for whatever we want! Any further hardware protection DOES NOT raise the end-user security but ONLY the "security" how software is protected FROM the end-user.
  Not only think about DRM and raising prices, think about vendor lock-in for Outlook, Office or any other application. Think about a global censor of people by just removing their public key (which would be required in case of trusted computing) from the network. I could go on…

To end this, Bruce Schneier said:

> *"There's a lot of good stuff in Palladium (alias NGSCB), and a lot I like about it. There's also a lot I don't like, and am scared of. My fear is that Palladium will lead us down a road where our computers are no longer our computers, but are instead owned by a variety of factions and companies all looking for a piece of our wallet. To the extent that Palladium facilitates that reality, it's bad for society. I don't mind companies selling, renting, or licensing things to*

> *me, but the loss of the power, reach, and flexibility of the computer is too great a price to pay."*

And finally Bill Gates:

> *"We came at this thinking about music, but then we realized that e-mail and documents were far more interesting domains."*

For further reading (about DRM, NGSCB, etc.), look at the following sites:

- http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html (recommended)
- http://www.microsoft.com/technet/archive/security/news/ngscb.mspx?mfr=true
- http://en.wikipedia.org/wiki/Next-Generation_Secure_Computing_Base
- http://www.cs.bham.ac.uk/~mdr/teaching/TrustedComputing.html

If you want awesome technical articles about how PatchGuard is working, I can only recommend the following (mine is about how to disable it and not that much about how PatchGuard is working):

- PatchGuard 1: http://www.uninformed.org/?v=3&a=3&t=pdf (recommended)
- PatchGuard 2: http://www.uninformed.org/?v=6&a=1&t=pdf
- PatchGuard 3: http://www.uninformed.org/?v=8&a=5&t=pdf

There is another thing to mention. You can of course disable PatchGuard in a DOCUMENTED, STABLE and EASY manner, by running the following commands in a root-shell and restarting the PC afterwards:

```
Bcdedit /debug ON
Bcdedit /dbgsettings SERIAL DEBUGPORT:1 BAUDRATE:115200 /start AUTOENABLE /noumex
```

"noumex" will disable user mode exceptions for kernel debuggers which in fact would Visual Studio prevent from working. "AUTOENABLE" will force PatchGuard to be disabled, because even if you don't attach a kernel debugger, you could do it at any time, and that is enough. Don't use this setting to write kernel patching software for end-users. The DEBUG switches will have many side-effects. Mainly with Visual Studio and probably other debuggers, but also with DRM content playback, I suppose. I also noticed an annoying system slowdown and a huge overall latency. The boot time will be increased too, probably because Windows is waiting for a debugger…

Why is PatchGuard disabled with these settings? Simply, because to set breakpoints, you will have to overwrite kernel code, for example, with INT3 and that would already be enough for PatchGuard to BSOD. Another aspect is that a debugger is a common way to explore PatchGuard ;-).

1.1 A word of warning

This driver is NOT intended to be used in any end-user scenarios. It has been tested on Windows Vista x64 (all updates, 30.07.2008) and on Windows Vista x64 SP1 (all updates, 30.07.2008). It is known to not work on an out-dated Windows Vista, so make sure that all PatchGuard related updates (better all updates), released before the above dates, are installed. Any future Windows Update may shot this approach to hell, and would BSOD the systems of your customers in the worst case! This is not limited to my approach… There is no way to bypass PatchGuard on end-user PCs, but only on your own, where you have control about updates and may hide all future PatchGuard related ones, for example! The Symantec Showcase, how I like to call it, has proven that you only can rely on documented things, especially when dealing with the kernel. Like Microsoft said to Symantec, that they will release an update, BSODing related PCs, if Symantec puts PatchGuard bypassing code into their products, I am saying to you: Don't ever release a PatchGuard bypassing or 64-Bit undocumented kernel hooking product to customers! An option may be the PatchGuard API, but I think it is not publicly available. BTW I disagree to Symantec's opinion that they need to use undocumented kernel patching to develop security products. The only thing is that they already had such products and it would be very expensive for them to reinvent the wheel, by not using kernel patching. Microsoft just did the right thing by not listening to Symantec. Undocumented kernel patching can never be an option to raise security in a trusted environment like Vista 64-Bit. It will probably even lower security AND stability. It is Microsoft's duty to protect the kernel against malware and this will already work very well with UAC, enforced driver signing (it really was a burden to get my test driver installed the first time, so I think the kernel security performs quite well) and PatchGuard. The story that "malware is able to bypass PatchGuard" is something strange. I never saw a signed virus. And even if so, there is no way to do something against it if malware already is in the kernel. In this case you just lost the fight. Malware instead should NEVER get into the kernel; this is what security software should care about and not some kind of unstable, undocumented and also unsafe or even useless Post-Mortem-Security.

Table of Content

1.2 Goals of PatchGuard

The following data and structures are protected by PatchGuard:

- Modifying system service tables, for example, by hooking *KeServiceDescriptorTable*
- Modifying the interrupt descriptor table (IDT)
- Modifying the global descriptor table (GDT)
- Using kernel stacks that are not allocated by the kernel
- Patching any part of the kernel (detected only on AMD64-based systems)

PG uses a system check routine that validates all the above structures and data. Basically PatchGuard is about how to protect this system check routine from being cracked. Cracking means to violate at least one of the above statements without knowledge of PatchGuard. This article describes how to violate ALL of them by completely disabling PatchGuard!

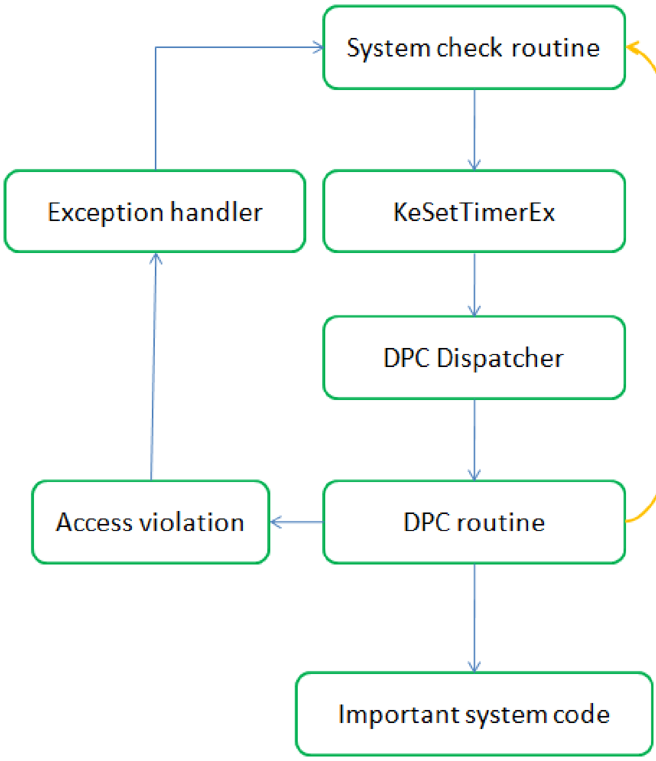
In the following I will refer to so called “code paths”. Those are meant to invoke the system check routine. The simplest way is to directly call it, like you would call any other API. But PatchGuard is about making it as hard as possible to

- Find out how the system check routine is invoked
- Interfere with this invocation (by preventing/redirecting it)
- Disassemble the system check routine or find out its entry point address

To accomplish this task, PatchGuard uses various tricks and uncommon processor behavior to obfuscate as much as possible. The system check routine itself will be executed every few minutes.

1.3 A brief overview

The following is a diagram for the code logic of PatchGuard:



The orange arrow applies to PatchGuard 3 only. As you can see, PatchGuard uses the general purpose mechanism for delayed code execution (timers). There are ten PatchGuard related DPC routines:

CmpEnableLazyFlushDpcRoutine
CmpLazyFlushDpcRoutine
ExpTimeRefreshDpcRoutine
ExpTimeZoneDpcRoutine
ExpCenturyDpcRoutine
ExpTimerDpcRoutine
IopTimerDispatch
IopIrpStackProfilerTimer
KiScanReadyQueues
PopThermalZoneDpc

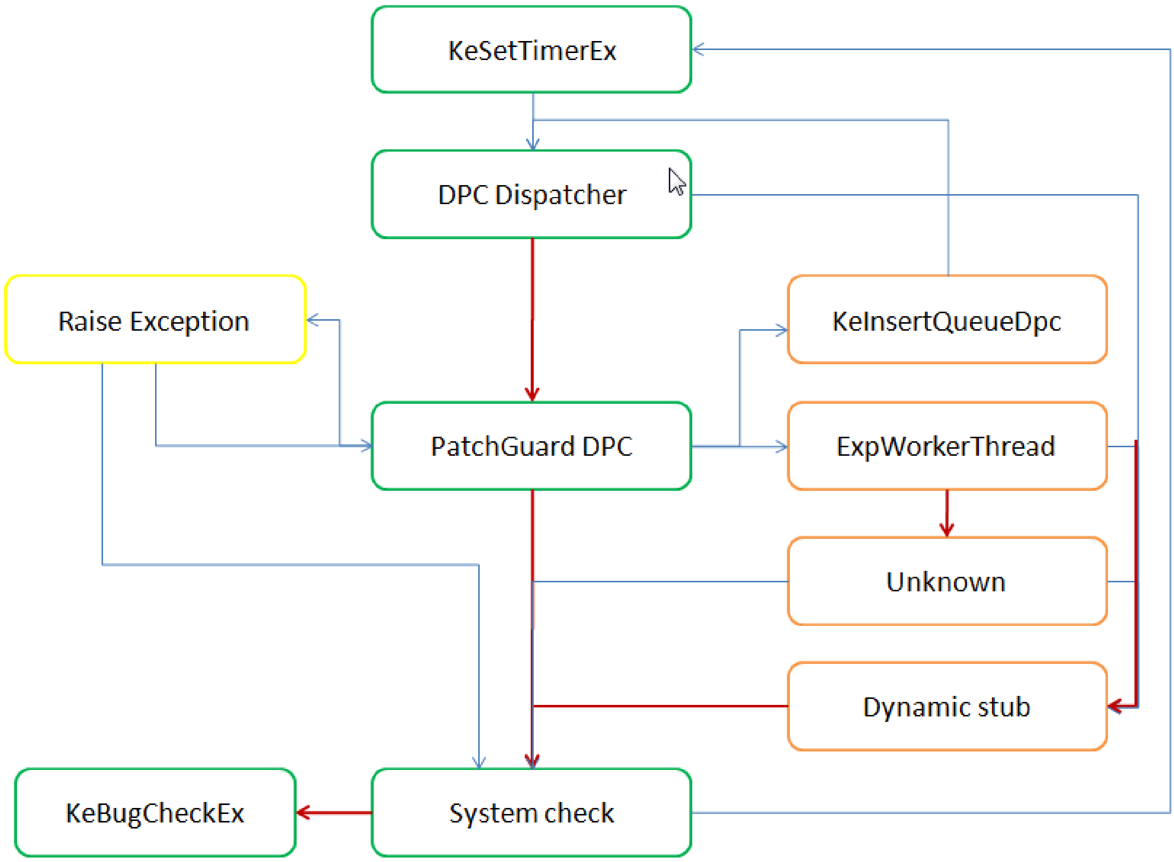
But keep in mind that those are also executing important system code if PatchGuard is not invoked. This switch is made by passing an invalid pointer as *DeferredContext*. One may say that this is buggy, because dereferencing an invalid pointer at dispatch level throws a non-catchable trap resulting in a bug check. But PatchGuard is using a so called non-canonical pointer. Such a pointer does not follow the x64 processor specification that requires a pointer to have to upper 16 bits either set to one or zero (that is 0x0000 or 0xFFFF). A non-canonical pointer starts with 0x6238, 0xF10A and so on.

Dereferencing them will instead cause a general protection fault, which is catchable and in case of PatchGuard, executes the system check routine.

PatchGuard 3 further has the ability to resume execution in the DPC routine after a #GP, probably raising another one, which again could decide to invoke the system check routine or continue execution. PatchGuard 3 also may execute the system check routine without raising an exception at all. What it does for a particular DPC depends on the encrypted *DeferredContext* and probably also on the DPC itself. So we just don't know!

1.4 Code flow diagram

The following is a code flow diagram of PatchGuard. The green boxes apply to all PatchGuard versions whereas the yellow box only applies since PatchGuard 2 and the orange boxes to PatchGuard 3. Please keep in mind that most of the stuff is just "guessing". I am optimistic that the following diagram will be a good approximation:



As you can see, PG3 has become much more flexible and thus much harder to bypass. My driver will attack the red arrows:

“DPC Dispatcher” to “PatchGuard DPC”

The driver filters all DPCs with PatchGuard specific parameters as *DeferredContext*. Further it catches the non-SEH code path of PatchGuard which previously has been overwritten with unhandled breakpoints, finding their way back to the exception handler in my driver! Of course there are still some DPCs left and this is why we need to add further blocking.

“Dynamic stub” to “System check” and “PatchGuard DPC” to “System check”

By overwriting the dynamic stubs and parts of the PatchGuard DPCs with breakpoints, execution continues in the DPC interceptor’s exception handler instead of the system check routine.

“ExpWorkerThread” to “Dynamic stub” and “Unknown”

Another hook is applied to the worker thread queue. This will filter all dynamic worker routines and wrap all others in a try-except statement. Some special kinds of system check invocations which seem to be incompatible with the next blocking mechanism are always originated from the worker queue. But as I said, we just filter them, so their incompatibility won’t cause any harm…

“System check” to “KeBugCheckEx”

Only PatchGuard methods raised over *ExQueueWorkItem* get here. It is a burden to reproduce this case because you have to restart 10 – 30 times…

This block just suspends all *CRITICAL_STRUCTURE_CORRUPTIONs*. So they never cause the system to BSOD.

The problem was that the driver caught the bug check attempt but failed to suspend the calling worker thread. Then I also hooked *ExpWorkerThread* and since that step, only suspend able calls to *KeBugCheckEx* are made.

1.5 Some call stacks

By patching some of the code paths, I could extract interesting call stacks on their way to the system check routine.

The following is one of the longest possible call stacks in case of PatchGuard. It raises two handled #GPs and finally invokes the system check routine directly from the DPC without exception handling:

```
nt!KeBugCheckEx
nt! ?? ::FNODOBFM::`string'+0x12767
nt!KiExceptionDispatch+0xae
nt!KiBreakpointTrap+0xb7
nt!ExpTimeRefreshDpcRoutine+0x1e6
nt!_C_specific_handler+0x8c
nt!RtlpExecuteHandlerForException+0xd
```

```
nt!RtlDispatchException+0x228
nt!KiDispatchException+0xc2
nt!KiExceptionDispatch+0xae
nt!KiGeneralProtectionFault+0xcd
nt!ExpTimeRefreshDpcRoutine+0xf1
nt!_C_specific_handler+0x140
nt!RtlpExecuteHandlerForUnwind+0xd
nt!RtlUnwindEx+0x233
nt!_C_specific_handler+0xcc
nt!RtlpExecuteHandlerForException+0xd
nt!RtlDispatchException+0x228
nt!KiDispatchException+0xc2
nt!KiExceptionDispatch+0xae
nt!KiGeneralProtectionFault+0xcd
nt!KiCustomRecurseRoutine0+0xd
nt!KiCustomRecurseRoutine9+0xd
nt!KiCustomRecurseRoutine8+0xd
nt!KiCustomRecurseRoutine7+0xd
nt!KiCustomAccessRoutine7+0x22
nt!ExpTimeRefreshDpcRoutine+0x54 ← PatchGuard DPC
nt!KiRetireDpcList+0x155
nt!KiIdleLoop+0x5f
nt!KiSystemStartup+0x1d4
```

The interesting thing about this one is that the #GPs were handled by PatchGuard, but didn't invoke the system check routine. So we have proven that an access violation is not necessarily invoking the SCR since PatchGuard 3.

The next one in contrast is one of the shortest possible call stacks. It invokes the system check routine over a resident method (copied into non-paged pool). You will probably see where this article with go to, because my driver is also listed in the call stack:

```
nt!KeBugCheckEx
nt! ?? ::FNODOBFM::`string'+0x12767
nt!KiExceptionDispatch+0xae
nt!KiBreakpointTrap+0xb7
0xfffffa80010b3cc9 ← PatchGuard's dynamic method
0xfffffa80010bbc00 ← PatchGuard's optional intro
PG3Disable!VistaAll_DpcInterceptor+0x34
nt!KiRetireDpcList+0x117
nt!KiIdleLoop+0x62
…
```

Someone who already tried to disable PatchGuard will probably wonder how I got these call stacks. The *KiBreakPointTrap* should be self explaining. I patched the code paths with unhandled breakpoints. So instead of invoking the system check routine, *KeBugCheckEx* will create a memory dump and we can work with it in a post-mortem session using WinDbg.

1.6 Detecting non-canonical, pseudo random contexts

One way to filter PatchGuard DPCs was to detect its special *DeferredContext* values and cancel related timers. Since PatchGuard 3, this alone is not sufficient enough. But it is still a good starting point to filter out most of PatchGuard's DPCs before they actually raise the system check routine.

The following code snipped is able to tell whether a given pointer is a PatchGuard context or not:

```
BOOLEAN CheckSubValue(ULONGLONG InValue)
{
	ULONG		i;
	ULONG		Result;
	UCHAR*		Chars = (UCHAR*)&InValue;

	// random values will have a result around 120...
	Result = 0;

	for(i = 0; i < 8; i++)
	{
		Result += ((Chars[i] & 0xF0) >> 4) + (Chars[i] & 0x0F);
	}

	// the maximum value is 240, so this should be safe...
	if(Result < 70)
		return TRUE;

	return FALSE;
}

BOOLEAN PgIsPatchGuardContext(void* Ptr)
{
	ULONGLONG	Value = (ULONGLONG)Ptr;
	UCHAR*		Chars = (UCHAR*)&Value;
	LONG		i;

	// those are sufficient proves for canonical pointers...
	if((Value & 0xFFFF000000000000) == 0xFFFF000000000000)
		return FALSE;

	if((Value & 0xFFFF000000000000) == 0)
		return FALSE;


	// sieve out other common values...
	if(CheckSubValue(Value) || CheckSubValue(~Value))
		return FALSE;

	if(Ptr == NULL)
		return FALSE;

	//This must be the last check and filters latin-char UTF16 strings...
	for(i = 7; i >= 0; i -= 2)
	{
		if(Chars[i] != 0)
			return TRUE;
	}
```

```
    // this should only return true if the pointer is a unicode string!!!
    return FALSE;
}
```

The problem is that our task here is NOT to filter non-canonical pointers. Our task is to distinguish between PatchGuard context parameters and any other common *DeferredContext* value such as zero, some table indices like "1, 2, 3, 4, 5, …", Unicode sequences, etc. All in all the above code detects pseudo random values.

With this ability in mind, we can now look at how to patch the DPC dispatcher, which is the only way to apply these custom checks.

2 Disarming PatchGuard 3

In order to disable PatchGuard 3, we will have to block all DPCs with a PatchGuard specific context and to catch the exceptions raised by unhandled breakpoints. But there still seem to be code paths left, running in a worker queue, executing the system check routine and finally raising the bug check. For this purpose we will also hook *KeBugCheckEx* and suspend all threads raising a *CRITICAL_STRUCTURE_CORRUPTION*. Please note that the latter is a very rare case and won't impact system performance. But this is still not enough. Some of the *KeBugCheckEx* calls are not suspend able. I solved this by also hooking *ExpWorkerThread,* filtering out dynamic stubs and wrapping all others in a try-except statement.

2.1 Finding the Windows DPC invocation code

It is time to mention that *KiRetireDpcList* and *KiTimerExpiration* are the only points in the kernel which are responsible for dispatching queued DPCs. If you look at the disassembly for *KiTimerExpiration* and *KiRetireDpcList*, which is quite too long for showing, you will find the following four indirect call code blocks:

```
nt!KiTimerExpiration+0x888:
        488b5308        mov     rdx, qword ptr [rbx+8]
        488b4bf8        mov     rcx, qword ptr [rbx-8]
        4d8bcc          mov     r9, r12
        4c8bc7          mov     r8, rdi
        ff13            call    qword ptr [rbx]
        4084f6          test    sil, sil
        742c            je      nt!KiTimerExpiration+0x8d3
```

```
nt!KiTimerExpiration+0x679:
        488b5308        mov     rdx,qword ptr [rbx+8]
```

```
        488b4bf8        mov     rcx,qword ptr [rbx-8]
        4d8bcc          mov     r9,r12
        4d8bc5          mov     r8,r13
        ff13            call    qword ptr [rbx]
        4084ed          test    bpl, bpl
        742c            je      nt!KiTimerExpiration+0x7e5

    nt!KiTimerExpiration+0x799:
        488b5308        mov     rdx,qword ptr [rbx+8]
        488b4bf8        mov     rcx,qword ptr [rbx-8]
        4d8bcc          mov     r9,r12
        4d8bc5          mov     r8,r13
        ff13            call    qword ptr [rbx]
        4084ed          test    bpl, bpl
        742c            je      nt!KiTimerExpiration+0x7e5

    nt!KiRetireDpcList+0x145:
        4d8bcc          mov     r9, r12
        4c8bc5          mov     r8, rbp
        488bd6          mov     rdx, rsi
        488bcf          mov      rcx, rdi
        ff542470        call    qword ptr [rsp+70h]
        4584ff          test    r15b, r15b
        742b            je      nt!KiRetireDpcList+0x185
```

What we can see is that they are very similar and in all tested kernel images, these four code blocks were unique. So what are those blocks actually doing? They will invoke the user defined DPC routine…

But how does the thing look like under Windows Vista SP1? *KiTimerDispatch* seems to be disappeared. The following are the only two points in the SP1 kernel, executing user DPCs:

```
    Nt!KiTimerListExpire+0x31a:
        458b4e04            mov     r9d,dword ptr [r14+4]
        458b06              mov     r8d,dword ptr [r14]
        4189ac24a0370000    mov     dword ptr [r12+37A0h], ebp
        488b5308            mov     rdx, qword ptr [rbx+8]
        488b4bf8            mov     rcx, qword ptr [rbx-8]
        ff13                call    qword ptr [rbx]
        4084ff              test    dil, dil
        0f856c8ffdff        jne     nt! ?? ::FNODOBFM::`string'+0x39742

    nt!KiRetireDpcList+0x107:
        4d8bce              mov     r9,r14
        4d8bc5              mov     r8,r13
        498bd4              mov     rdx,r12
        488bcb              mov     rcx, rbx
```

```
    ff542470        call    qword ptr [rsp+70h]
    4584ff          test    r15b,r15b
    0f856e7ffdff    jne     nt! ?? ::FNODOBFM::`string'+0x39888
```

One will observe, that even if the whole underlying code logic has changed (only two DPC invocations, another method for expiring timers, far jumps at the end, etc.), it does still look very similar. Also the chance that a usual Windows Update changes the machine code for this part is quite small. Before developing any flexible mechanism we just search for the plain bytes. This works and is much more stable. For every update that changes the bytes we can simply add additional search vectors, currently I only know those two…

2.2 Patching the Windows DPC invocation code

From now on I focus on Service Pack 1, because it only has two methods to patch and is more up to date. The code without Service Pack is quite similar and you should have no problems to understand it, once you got it for SP1.

How we can patch *KiTimerExpiration* and *KiRetireDpcList* if we nowhere can insert a JMP instruction without messing up the whole code logic? We will overwrite the last MOV instruction too and emulate it in a proper jumper table before we continue execution in our interception method:

```
Nt!KiTimerListExpire+0x31a:
    458b4e04            mov     r9d,dword ptr [r14+4]
    458b06              mov     r8d,dword ptr [r14]
    4189ac24a0370000    mov     dword ptr [r12+37A0h], ebp
    488b5308            mov     rdx, qword ptr [rbx+8]
    90                  nop
    E8XXXXXXXX          call    TIMER_FIX
    4084ff              test    dil, dil
    0f856c8ffdff        jne     nt! ?? ::FNODOBFM::`string'+0x39742
```

```
nt!KiRetireDpcList+0x107:
    4d8bce              mov     r9,r14
    4d8bc5              mov     r8,r13
    498bd4              mov     rdx,r12
    90                  nop
    90                  nop
    E8XXXXXXXX          call    DPC_FIX
    4584ff              test    r15b,r15b
    0f856e7ffdff        jne     nt! ?? ::FNODOBFM::`string'+0x39888
```

Remember that this operation can and has to be performed atomically, by using a 64-Bit wide MOV instruction!

Some of you may probably ask their selves, where the hell do we wanna jump to with a 32-Bit offset and the paged and non-pages pools being terabytes away?! I also thought much about it and there are only two solutions. The first one is a sledgehammer approach and I don't like such. This is by stopping all CPUs except one, raising IRQL to *HIGH_LEVEL* and putting an absolute call instruction in the two code blocks above. This is the only way one could replace more than eight bytes as an atomic operation. The other way is to hijack one of those ten *KiCustomAccessRoutines*. So what we do here is placing a jumper as first instruction of such a routine and redirecting it to another one. Now we can work with the other bytes, and there are quite enough, to build our jump table. Of course I know that even the atomic MOV instruction is not 100% safe. But the chance that a thread will be between the MOV and CALL, which our patch overwrites, and all this within such a short timeframe, is quite small.

The jump table has to do two things. Firstly it should recover the overwritten MOV instruction and then invoke our interception method using an absolute far jumper. In case of SP1 we have to take care of two different cases and the code we place in such a custom access routine may look like this:

```
nt!KiCustomAccessRoutine4:
        E9XXXXXXXX              Jmp         KiCustomAccessRoutine0

TIMER_FIX:
        488b4bf8                mov         rcx,qword ptr [rbx-8]
        eb03                    jmp         INTERCEPTOR

DPC_FIX:
        488bcb                  mov         rcx,rbx

INTERCEPTOR:
        48b8XXXXXXXXXXXXXXXX    mov         rax, VistaAll_DpcInterceptor
        ffe0                    jmp         rax
```

Well, that's the entire mystic about patching the DPC invocation code. Now ANY DPC is going through our interceptor. You probably can imagine that there are a lot, because most interrupts will also raise a DPC; thus, the interceptor has to be executed as fast as possible.

2.3 Designing the interceptor

Now we should take a look at the DPC interceptor:

```
void VistaAll_DpcInterceptor(
            PKDPC InDpc,
            PVOID InDeferredContext,
            PVOID InSystemArgument1,
            PVOID InSystemArgument2)
{
      ULONGLONG         Routine = (ULONGLONG)InDpc->DeferredRoutine;

      __try
```

```
        {
                if((Routine >= 0xFFFFFA8000000000) &&
                        (Routine <= 0xFFFFFAA000000000))
                {
                }
                else if(KeContainsSymbol((void*)Routine))
                {
                        if(!PgIsPatchGuardContext(InDeferredContext))
                                InDpc->DeferredRoutine(
                                        InDpc,
                                        InDeferredContext,
                                        InSystemArgument1,
                                        InSystemArgument2);
                }
                else
                        InDpc->DeferredRoutine(
                                InDpc,
                                InDeferredContext,
                                InSystemArgument1,
                                InSystemArgument2);
        }
        __except(EXCEPTION_EXECUTE_HANDLER)
        {
        }
}
```

The first thing we check is whether the DPC routine in either in the confines of the kernel image, where all of the ten DPCs reside, or in the memory pool, where the dynamic invocation stubs reside. This way we can reduce the chance that we are cancelling non-PatchGuard DPCs, because the kernel is very unlikely to have pseudo random numbers as *DeferredContext*. Also I don't know any sane driver that needs dynamic DPCs. After this check we can just skip all obvious PatchGuard DPCs and *DeferredRoutines* residing in dynamic memory.

You may wonder about the exception frame. I mentioned earlier, that PatchGuard 3 introduces a code path that is not using exceptions and also only canonical *DeferredContexts*. This way it would pass our filter and get to *KeBugCheckEx*. The problem now is that PatchGuard may decide to directly invoke it (not using a worker thread), thus our hook would run at DPC level and cause a bug check. I solved this by overwriting some fingerprints with breakpoints, which actually converts such non-SEH code paths into SEH ones, because unhandled breakpoints throw a catchable exception! The following is the prototype of all memory resident dynamic methods:

```
nt!KiTimerDispatch:
        6690                    xchg    ax,ax
        9c                      pushfq
        4883ec20                sub     rsp,20h
        8b442420                mov     eax,dword ptr [rsp+20h]
        4533c9                  xor     r9d,r9d
        4533c0                  xor     r8d,r8d
        4889442430              mov     qword ptr [rsp+30h],rax
        488b4140                mov     rax,qword ptr [rcx+40h]
        48b90000000000f8ffff    mov     mov rcx,0FFFFF80000000000h
        4833c2                  xor     rax,rdx
        480bc1                  or      rax,rcx
```

48b9f048311148315108	**mov**	**mov rcx,8513148113148F0h**
488b10	mov	rdx,qword ptr [rax]
c700f0483111	mov	dword ptr [rax],113148F0h
4833d1	xor	rdx,rcx
488bc8	mov	rcx,rax
ffd0	call	rax
4883c420	add	rsp,20h
59	pop	rcx
c3	ret	

The driver searches for the bold printed byte sequence and overwrite it with breakpoints; that's all!

Please note that the exception frame doesn't bring any advantage for the SEH code path. I tried to catch it without the DPC filtering but the Driver Verifier does not seem to like it. Fortunately, the SEH code path does ALWAYS use non-canonical *DeferredContext* values and that's why we can filter it entirely.

2.4 Hooking ExpWorkerThread

Patching the DPC code alone is not enough. PatchGuard 3 also uses worker items to accomplish the same task. Our worker thread interceptor looks something similar, but is not the same:

```
VOID VistaAll_ExpWorkerThreadInterceptor(
            PWORKER_THREAD_ROUTINE InRoutine,
            VOID* InContext,
            VOID* InRSP)
{
      ULONGLONG         Val = (ULONGLONG)InRoutine;

    if((Val >= 0xfffffa8000000000) && (Val <= 0xfffffaa000000000))
            return;

    __try
    {
          InRoutine(InContext);
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
    }
}
```

What we do here is to filter out all routines residing in dynamic memory, just like we did it in the DPC hook. Additionally we wrap all other routines in a try-except statement. The context of a PatchGuard worker thread is canonical, so we have no chance to filter it out explicitly.

What I experienced so far is that PatchGuard is using dynamic methods also for the worker queue. After blocking them like above, there were still some bug checks left. They were originated from

ExpWorkerThread, but never gone through our handler! This is something strange because the disassembly shows that we only have on single point where the worker items are actually called. I just can imagine that PatchGuard is again using exceptions to redirect execution to a special exception handler, just like it did for the DPC handler. And this is why I also wrap the calls in a try-except statement!

The procedure of patching the worker queue is very similar to the DPC queue. We have the following bytes to patch:

```
nt!ExpWorkerThread+0x11a:
        5c                  pop       rsp
        2470                and       al,70h
        4c8b6318            mov       r12,qword ptr [rbx+18h]
        488b7b10            mov       rdi,qword ptr [rbx+10h]
        498bcc              mov       rcx,r12
        ffd7                call      rdi
        4c8d9ee0020000      lea       r11,[rsi+2E0h]
        4d391b              cmp       qword ptr [r11],r11
```

And redirect them to the very same jump table. The difference is that we don't restore the registers in the jump table but in a prepared jump target in our driver:

```
VistaSp0_ExpWorkerThread_Fix PROC

        mov             rcx, rdi
        mov             rdx, r12
        mov             r8, rsp
        jmp             VistaAll_ExpWorkerThreadInterceptor

VistaSp0_ExpWorkerThread_Fix ENDP
```

This will call the above interceptor and we are done!

2.5 Hooking KeBugCheckEx

With the work done so far, we are able to prevent, let's say, 95% of all system check routine invocations. The other 5% will pass our DPC filter and are still causing a BSOD. I said earlier that we are going to hook *KeBugCheckEx* to solve this issue. Well it is not that easy, because PatchGuard actually overwrites the method with a fresh copy, before invoking it. So we need to hook a subroutine. If we look at the first instructions

```
nt!KeBugCheckEx:
```

```
48894c2408              mov     qword ptr [rsp+8],rcx
4889542410              mov     qword ptr [rsp+10h],rdx
4c89442418              mov     qword ptr [rsp+18h],r8
4c894c2420              mov     qword ptr [rsp+20h],r9
9c                      pushfq
4883ec30                sub     rsp,30h
fa                      cli
65488b0c2520000000      mov     rcx,qword ptr gs:[20h]
4881c120010000          add     rcx,120h
e8c1050000              call    nt!RtlCaptureContext
```

, we see that *RtlCaptureContext*() seems to be perfect for our task. This is because *KeBugCheckEx*() is calling it very early and thus the system is still in a sane state. In order to hook *RtlCaptureContext*, we need to place a jumper as usual:

```
Nt!RtlCaptureContext:

        50              push    rax
        48b9Xxx         mov     rax, PG3Disable! RtlCaptureContext_Hook
        ffe0            jmp     rax
```

The difference is that we are hooking a context capturing method and thus have to ensure that this context in particular is NOT changed by our hook. This is why we have to backup volatile registers. The jumper will continue execution in a native interception stub which was built only for the purpose of hooking *RtlCaptureContext*:

```
RtlCaptureContext_Hook PROC

        ; call high level handler without messing up the context structure...
        pushfq
        push    rcx
        push    rdx
        push    r8
        push    r9
        push    r10
        push    r11
        mov     rcx, qword ptr[rsp + 136]
        mov     rdx, qword ptr[rsp + 8 * 8]
        sub     rsp, 32
        call    KeBugCheck_Hook
        mov     qword ptr [rsp], rax
        add     rsp, 32
        pop     r11
        pop     r10
        pop     r9
```

```
        pop             r8
        pop             rdx
        pop             rcx
        popfq
        pop             rax

        ; recover destroyed bytes of RtlCaptureContext
        pushfq
        mov             word ptr [rcx+38h],cs
        mov             word ptr [rcx+3Ah],ds
        mov             word ptr [rcx+3Ch],es
        mov             word ptr [rcx+42h],ss

        ; jump behind destroyed bytes... (return value of KeBugCheck_Hook)
        jmp                     qword ptr[rsp - 32 - 8 * 7 + 8]

RtlCaptureContext_Hook ENDP
```

Firstly, it safely calls our *KeBugCheck_Hook*() method with the bug check code as first parameter and the caller of *RtlCaptureContext* as second one. Secondly, it recovers the overwritten instructions of *RtlCaptureContext* and finally continues execution behind the jumper.

If you now look at the *KeBugCheck_Hook*

```
ULONGLONG KeBugCheck_Hook(ULONGLONG InBugCode, ULONGLONG InCaller)
{
      FAST_MUTEX        WaitAlways;

      if((InCaller >= KeBugCheckEx_Sym) &&
            (InCaller <= KeBugCheckEx_Sym + 100))
      {
            if(InBugCode == CRITICAL_STRUCTURE_CORRUPTION)
            {
                  /*
                        Enable interrupts, resets the stack pointer and
                        calls PgBlockWorkerThread!
                  */
                  EnableInterrupts();

                  ExInitializeFastMutex(&WaitAlways);

                  ExAcquireFastMutex(&WaitAlways);

                  ExAcquireFastMutex(&WaitAlways);
            }
      }

      return RtlCaptureContext_Sym + 14;
}
```

, you may observe that it just checks whether the caller is *KeBugCheckEx* and the bug check code is *CRITICAL_STRUCTURE_CORRUPTION*. If that is the case, it reenables interrupts, and block the thread forever. We are only able to do this, because we eliminated the chance, that this bug check was raised though the DPC dispatcher and it definitely would be, if we skip our DPC filter! If the caller was any other symbol, we just execute *RtlCaptureContext*. I assume that every thread switch will call this method, so this is the next critical execution path of Windows which we are hooking…

I am sure someone will ask whether this doesn't lead to an endless thread creation loop with a period of some mintues. Again, we can only do this, because we disabled the DPC code path already. As the invokation method is randomly choosen, we cause one or two orphaned threads until PatchGuard is never executed again (because it does not always use *ExQueueWorkItem* and if we block all other code paths, there is a point when no PatchGuard contexts are left).

OPERATION SUCCEEDED, PATIENT DEAD

So what we have done so far? We disabled PatchGuard 3 on Windows Vista SP1, all updates installed. Of course the patches we applied were not that common coding style ;-). But everyone will agree that potential malware is written like that and actually the patches are very stable for a given OS. You may rollback all changes after approx. 20 minutes. The reason is that PatchGuard will only add a new code path, if the system check routine is invoked. So when we block its execution for a reasonable period of time, there is nothing to block anymore… My driver does not support rolling back the changes so far and in fact never will. This is also why you can't unload it after patching.

3 Using and compiling the drivers

If you never developed a driver for 64-Bit, you have several obstacles to solve. Therefore I prepared a clean Windows installation. Now I will describe how you can build my drivers and install them.

3.1 Preparing the build environment

As first step you should download the latest Windows Driver Kit from https://connect.microsoft.com/ and install it to "C:\WinDDK" for example. You have to register and sign in to Microsoft Passport I believe but after all it is still free.

Then you have to add a system wide environment variable called "WDKROOT" with the value of your installation path.

From now on I assume that you extracted the DisablePatchGuard archive to "C:\PGDisable"; so that the project solution is located in "C:\PGDisable\PGDisable.sln".

To install a driver on Windows Vista, you have to sign it. Now open a root-shell and type the following commands:

```
> Bcdedit –set TESTSIGNING ON
```

You need to restart the PC. After this we are ready to create a test certificate:

```
> cd c:\winddk\bin\SelfSign

> MakeCert -r -pe -ss PGDisableCertStore -n  "CN=PGDisableCert"
"c:\PGDisable\PGDisableCert.cer"
```

Now open the certificate in the Explorer by double clicking it. Currently the certificate is an untrusted one. To make Windows trusting the certificate, click “Install certificate” – “Next” – “place all certificates in the following store” – “Trusted Root Certification Authorities” – “OK” – “Next” – “Finish” and do the same procedure to add it to the “Trusted Publishers” store.

The project comes with a post-build event that relies on the fact that a certificate named “PGDisableCert” is in the “PGDisableCertStore”. This will automatically sign the drivers after each build and you have to care about nothing.

Now we are done and you can start building the project!

3.2 Service management

To keep the code simple, I didn’t provide much service management functionality. The application checks whether the driver is already installed. If not, it installs the driver. If yes, it just ensures that it is running.

But if you want to frequently recompile it, you need to make sure that the driver has been unloaded and removed from the service control manager, before you restart the application. Otherwise your newly compiled driver won’t be loaded because the old one is already there…

If you want to remove the driver from the service manager, just type “sc delete PG3Disable.sys” into a root shell. If the driver is currently running, this command will still succeed, but the driver actually stays installed. To prevent this you also have to stop it by using “sc stop PG3Disable.sys”. But keep in mind that due to the missing rollback of patches you won’t be able to load the driver again in the current system session (if you disabled PatchGuard)!

During development a frequent system restart is common. This is why I develop such drivers in a virtual machine.

3.3 The driver interface

The drivers in fact are very easy to use. The following table shows the supported control codes for the PG3Disable-Driver:

IOCTL_PATCHGUARD_DUMP

Writes a list of fingerprints to "C:\patchguard.log". If you execute this command after disabling PatchGuard, the file should only contain eight custom access routines.

IOCTRL_PATCHGUARD_DISABLE

This silently disables PatchGuard on success.

Both control codes have no parameters and not return value. The only thing is the status code available through *GetLastError*:

ERROR_SUCCESS

Operation has been completed successfully.

ERROR_NOT_SUPPORTED

This is returned either in case your system is not supported or PatchGuard was already patched by the driver.

All other error values are not explicitly raised by my code but may be returned by invoked kernel APIs.

The PG2Disable-Driver works very similar but there are two main differences. Firstly, you can disable PatchGuard multiple times and always get *ERROR_SUCCESS* as result. Secondly, you may load/unload the driver as often as you like. Furthermore it exports an additional command:

IOCTL_PATCHGUARD_PROBE

Installs a test hook for *KeCancelTimer*. Please note that your system will BSOD if PatchGuard is not already disabled!

Please note that PG2Disable won't work on Windows Vista SP1.

4 Ideas for PatchGuard 4

I really appreciate what Microsoft has done so far in order to harden PatchGuard. I don't think that it takes much effort to raise the bar of work, necessary to disable PatchGuard, to a degree that can be considered as non-exploitable. I would eliminate the DPC dispatcher as the main point of failure. Microsoft should make critical symbols invisible through the debugger. This could possibly be done by exporting proper debugging APIs over service interrupts which are using all required symbols internally; this way the debugger doesn't need to know about them but can still keep all functionality. Also some sort of a private way for delayed code execution would improve the whole thing.

A practical approach to realize this idea would be to define a macro named "PATCHGUARD_INVOKATION", for example, and use it all over the windows source code, but only for unexported APIs of higher order (not called inside any public API). Then a pre-build event could automatically replace such a macro with randomly generated invocation stubs, or even do nothing for most occurrences. One could base the randomness on a constant value, so that major updates will use different constant values resulting in totally different PatchGuard invocation stubs, while minor updates won't cause any changes to PatchGuard related sections in the binaries.

The form of delayed execution could be further improved by adding additional code to the internal interrupt handlers and only invoke PatchGuard after a counter has been incremented X times. Then this code again can check in a more expensive way whether it is "time" to do a system check or not. Even if one can still fingerprint many parts of PatchGuard, one could not disable it in a stable manner, because there are always some parts missing.

Further, multiple versions of the system check routine, which then of course shall go through a code morphing engine, would do their part.

Remember, the goal is not to make PatchGuard unexploitable on a single machine. The goal is to prevent malware from disabling it in an automated manner on a wide range of machines.

5 Windows timer internals

Now I want to write a little bit about my first investigations of PatchGuard 3. At the beginning I thought I could just cancel all the PatchGuard timers and I'd be done. As you can see I was mistaking, but that's what programming is about ;-).

I don't want to connect this with disabling PatchGuard. The reason is that PG3 actually is not exploitable with a timer cancelling approach. PG2 in fact is and the PG2Disable-Driver has all of the code required to do this and also shows how to extract the unexported kernel symbols in a stable manner (not working since Service Pack 1; here you'd have to use fingerprinting like it is done for *KiRetireDpcList*, for example).

On the surface, there was nothing in the net that showed how to enumerate timers. Also the WDK documentation contained nothing about it. So I finally disassembled some of the timer routines like *KeSetTimerEx*, *KeCancelTimer*, etc. *KeCancelTimer* seems to be the best starting point, because it is so small:

```
// push     rbx
// sub      rsp,20h
KIRQL                       OldIrql = 0;
BOOLEAN                     Existed = FALSE;
PKSPIN_LOCK_QUEUE           LockArray = NULL;
```

```
ULONG                       LockIndex = 0;
KTIMER_TABLE_ENTRY*         TimerEntry = NULL;

// mov      r9,rcx
// call     nt!KiAcquireDispatcherLockRaiseToSynch
OldIrql = KiAcquireDispatcherLockRaiseToSynch();

// mov      bl,byte ptr [r9+3]
// test     bl,bl
// mov      r10b,al
Existed = InTimer->Header.Inserted;

// je       nt!KeCancelTimer+0x76
if(Existed)
{
// nt!KeCancelTimer+0x19:
      // mov      rcx,qword ptr gs:[28h]
      LockArray = KeGetPcr()->LockArray;

      // movzx    r8d, byte ptr [r9+2]
      // mov      eax,r8d
      // shr      eax,4
      // and      eax,0Fh
      // add      eax,11h

      LockIndex  =  ((InTimer->Header.Hand  /  sizeof(KSPIN_LOCK_QUEUE))  &
0x0F) + LockQueueTimerTableLock;

      // shl      rax,4
      // add      rcx,rax
      // call     nt!KeAcquireQueuedSpinLockAtDpcLevel
      KeAcquireQueuedSpinLockAtDpcLevel(&LockArray[LockIndex]);

      // mov      byte ptr [r9+3],0
      InTimer->Header.Inserted = FALSE;

      // mov      rax,qword ptr [r9+28h]
      // mov      rdx,qword ptr [r9+20h]
      // cmp      rdx,rax
      // mov      qword ptr [rax],rdx
      // mov      qword ptr [rdx+8],rax

      //jne      nt!KeCancelTimer+0x71
      if(RemoveEntryList(&InTimer->TimerListEntry))
      {
      //nt!KeCancelTimer+0x58:
            // lea      rdx,[r8+r8*2]
            // lea      rax,[nt!KiTimerTableListHead]
            // lea      r8,[rax+rdx*8]
            TimerEntry = &KiTimerTableListHead[InTimer->Header.Hand];

            // cmp      r8,qword ptr [r8]
            //jne      nt!KeCancelTimer+0x71
            if(TimerEntry == (KTIMER_TABLE_ENTRY*)TimerEntry->Entry.Flink)
            {
            //nt!KeCancelTimer+0x6c:
                  // or       dword ptr [r8+14h],0FFFFFFFFh
                  TimerEntry->Time.HighPart = 0xFFFFFFFF;
            }
      }

//nt!KeCancelTimer+0x71:
      //call     nt!KeReleaseQueuedSpinLockFromDpcLevel
```

```
      KeReleaseQueuedSpinLockFromDpcLevel(&LockArray[LockIndex]);
}
//nt!KeCancelTimer+0x76:
//call    nt!KiReleaseDispatcherLockFromSynchLevel
KiReleaseDispatcherLockFromSynchLevel();

// mov      cl,r10b
// call     nt!KiExitDispatcher
KiExitDispatcher(OldIrql);

// mov      al,bl
return Existed;

// add      rsp,20h
// pop      rbx
// ret
```

I already inserted the source code guessed from the disassembly. I don't want to explain how you get it, because this is mainly based on experience with compiler building and (dis-)assemblers.

As you can see, the C-Code for it is quite straightforward. Though, there is a bug in the timer management, but I will come back to that later. The following is a little documentation about totally undocumented and unexported kernel symbols.

KIRQL **KiAcquireDispatcherLockRaiseToSynch**()

Probably locks the timer and/or DPC database. Raises the IRQL to *DISPATCH_LEVEL* and returns the previous state.

void **KeAcquireQueuedSpinLockAtDpcLevel**(PKSPIN_LOCK_QUEUE)

Is similar to the publicly available *KeAcquireInStackQueuedSpinLock*. You may use this method to acquire any of the locks in *KPCR::LockArray*. Please note that this method shall be called at *DISPATCH_LEVEL* only. The following constants may be helpful to index the right one:

LockQueueDispatcherLock	*0*
LockQueueExpansionLock	*1*
LockQueuePfnLock	*2*
LockQueueSystemSpaceLock	*3*
LockQueueVacbLock	*4*
LockQueueMasterLock	*5*
LockQueueNonPagedPoolLock	*6*
LockQueueIoCancelLock	*7*
LockQueueWorkQueueLock	*8*
LockQueueIoVpbLock	*9*
LockQueueIoDatabaseLock	*10*
LockQueueIoCompletionLock	*11*
LockQueueNtfsStructLock	*12*
LockQueueAfdWorkQueueLock	*13*
LockQueueBcbLock	*14*
LockQueueMmNonPagedPoolLock	*15*
LockQueueTimerTableLock	*17*

void **KeReleaseQueuedSpinLockFromDpcLevel** (*PKSPIN_LOCK_QUEUE*)

Is similar to the publicly available *KeReleaseInStackQueuedSpinLock*. You have to call it to release any of the previously acquired locks in *KPCR::LockArray*. Please note that this method shall be called at *DISPATCH_LEVEL* only.

void **KiReleaseDispatcherLockFromSynchLevel**()

Probably releases the timer/DPC lock at *DISPATCH_LEVEL* and does NOT change the IRQL.

void **KiExitDispatcher**(KIRQL InOldIrql)

Shall be called at *DISPATCH_LEVEL* and will lower the IRQL to *InOldIrql*. I think this method was not combined with the previous method, to allow performing more operations at *DISPATCH_LEVEL* (without holding the lock) before actually lowering the IRQL. I recently read in some Microsoft paper, that it also may schedule a new thread, now after the DPC level operation is done…

With this information in mind, we are ready to build our own enumeration method:

```
// a little helper…
PKSPIN_LOCK_QUEUE KeTimerIndexToLockQueue(UCHAR InTimerIndex)
{
	return &(KeGetPcr()->LockArray[((InTimerIndex / 
sizeof(KSPIN_LOCK_QUEUE)) & 0x0F) + LockQueueTimerTableLock]);
}

// this is where the enumeration starts
OldIrql = KiAcquireDispatcherLockRaiseToSynch();

for(Index = 0; Index < TIMER_TABLE_SIZE; Index++)
{
	LockQueue = KeTimerIndexToLockQueue((UCHAR)(Index & 0xFF));

	KeAcquireQueuedSpinLockAtDpcLevel(LockQueue);

	// now we can work with the timer list...
	TimerListHead = &KiTimerTableListHead[Index];
	TimerList = TimerListHead->Entry.Flink;

	while(TimerList != (PLIST_ENTRY)TimerListHead)
	{
		Timer  = CONTAINING_RECORD(TimerList, KTIMER, TimerListEntry);
		TimerList = TimerList->Flink;

		// TODO: work with the timer…
	}

	KeReleaseQueuedSpinLockFromDpcLevel(LockQueue);
}

KiReleaseDispatcherLockFromSynchLevel();

KiExitDispatcher(OldIrql);
```

If you now want to cancel a timer during enumeration, you could use the following code snippet:

```
// TODO: work with the timer…
Timer->Header.Inserted = FALSE;

if(RemoveEntryList(&Timer->TimerListEntry))
      TimerListHead->Time.HighPart = 0xFFFFFFFF;
```

Since PatchGuard 2, the timer DPCs are encrypted, so don't try to dereference the pointer. The PG2Disable-Driver shows you how to obtain the two internal decryption keys *KiWaitNever* and *KiWaitAlways*. With those symbols you may decrypt the *KDPC* pointer with the following code:

```
ULONGLONG         RDX = (ULONGLONG)Timer->Dpc;

RDX ^= InKiWaitNever;
RDX = _rotl64(RDX, *KiWaitNever & 0xFF);
RDX ^= (ULONGLONG)Timer;
RDX = _byteswap_uint64(RDX);
RDX ^= *KiWaitAlways;

return (KDPC*)RDX;
```

Now you are able to enumerate all Windows timers in a stable and interlocked way, just like the OS does it. Please note that you can't call *KeCancelTimer* during the enumeration as this would cause a deadlock! The PG2Disable-Driver may write all the timer information to a log file, even if we are running at DPC level during enumeration.

5.1 The windows timer bug

Now we can compare our both code parts. In the code of *KeCancelTimer* we had:

```
TimerEntry = &KiTimerTableListHead[InTimer->Header.Hand];
```

And in our enumeration:

```
TimerListHead = &KiTimerTableListHead[Index];
```

Well, they look equal on the surface. But the difference here is that "Index" in our enumeration actually has a range from zero to 511. This is because the public WDK constant *TIMER_TABLE_SIZE* has a value of 512. Now you might see the problem: *InTimer->Header.Hand* is only one byte wide according to the publicly available *DISPATCH_HEADER* structure. This causes Hand to overflow if the timer is placed in a linked list with an index greater than 255. This also explains the strange switch we extracted from *KeCancelTimer*, which again checks whether the linked list is empty, even if the use of *RemoveEntryList* already proved it. Redmon probably realized that there is something wrong and applied this workaround to make sure that only the timestamp of empty timer lists is reset.

But actually this seems to cause no big trouble at all. Just a funny thing that we discovered the reason of such a bug with plain reverse engineering.