MICROSOFT WINDOWS 8.1 KERNEL PATCH PROTECTION ANALYSIS

Authors: Mark Ermolov, Artem Shishkin
Research [at] ptsecurity [dot] com

Positive Research, 2014

Contents

1. Introduction

Kernel Patch Protection (also known as "patchguard") is a Windows mechanism designed to control the integrity of vital code and data structures used by the operating system. It was introduced in Windows 2003 x64 and has been constantly improved in further Windows versions. In this article we present a descriptive analysis of the patchguard for the latest Windows 8.1 x64 OS, and primarily focus on patchguard initialization and attack vectors related to it.

It is natural that kernel patch protection is being developed incrementally, so the initialization process is common for all versions of Windows that have patchguard. There are a lot of papers published about kernel patch protection on Windows, which describe the process of its initialization, so you may use references at the end of this article to obtain details.

2. Initialization sources

As widely known, the main component of patchguard is initialized in a misleadingly named function "KiFilterFiberContext". It will be the starting point of our investigation. Looking for cross-references doesn't help us much for pointing out its call site, but several articles help us by stating that patchguard initialization is called indirectly in a function "KeInitAmd64SpecificState". By indirectly we mean here not just an indirect call, but the usage of exception handlers. It is a very common trick often found in patchguard-related functions, as we'll see further. So, we have an initialization function call stack:

... --> Phase1InitializationDiscard --> KeInitAmd64SpecificState -/-> KiFilterFiberContext
(call) (call) (exception)

This type of initialization is described in more detail in [1]. By the way, this one is always called on the last CPU core, if it matters.

However, it is not the only way that kernel uses to initialize patchguard. With a 4% probability patchguard context can also be initialized from a function also misleadingly called "ExpLicenseWatchInitWorker":

... --> Phase1InitializationDiscard --> sub_14071815C (obviously with a stripped symbol because this one processes Windows license type for a current PC) --> ExpLicenseWatchInitWorker

The pseudocode of this function looks like this:

```
VOID ExpLicenseWatchInitWorker()
{
	PVOID KiFilterParam;
	NTSTATUS (*KiFilterFiberContext)(PVOID pFilterparam);
	BOOLEAN ForgetAboutPG;

	// KiServiceTablesLocked == KiFilterParam
	KiFilterParam = KiInitialPcr.Prcb.HalReserved[1];
	KiInitialPcr.Prcb.HalReserved[1] = NULL;

	KiFilterFiberContext = KiInitialPcr.Prcb.HalReserved[0];
	KiInitialPcr.Prcb.HalReserved[0] = NULL;

	ForgetAboutPG = (InitSafeBootMode != 0) | (KUSER_SHARED_DATA.KdDebuggerEnabled
	>> 1);
```

```
        // 96% of cases will fail
        if ( __rdtsc() % 100 > 3 )
                ForgetAboutPG |= 1;

        if ( !ForgetAboutPG && KiFilterFiberContext(KiFilterParam) != 1 )
                KeBugCheckEx(SYSTEM_LICENSE_VIOLATION, 0x42424242, 0xC000026A, 0, 0);
}
```

As you may notice, there is a small "present" in the "HalReserved" processor control block field left for this initialization case. Tracing down the guy who left it leads us to the very beginning of system startup:

... --> KiSystemStartup --> KiInitializeKernel --> KeCompactServiceTable --> KiLockServiceTable --> ??????

We have to pause here, because there is no code that puts data into HalReserved fields directly. As instead, it is done using the exception handler. And it is done in a different way from "KeInitAmd64SpecificState", because it doesn't trigger any exceptions. What it does instead is – it directly looks up the current instruction pointer, finds the corresponding function and it's exception handler manually, and then calls it. The exception handler of "KiLockServiceTable" function is an unnamed stub to the "KiFatalExceptionFilter".

?????? --> KiFatalExceptionFilter

"KiFatalExceptionFilter" in turn looks up an exception handler for "KiServiceTablesLocked" function. And surprisingly it is the "KiFilterFiberContext"! Also, a parameter that is passed to "KiFilterFiberContext" is located right after the "KiServiceTablesLocked" function. It is a small structure:

```
typedef struct _KI_FILTER_FIBER_PARAM
{
        NTSTATUS (*PsCreateSystemThread)();    // a pointer to PsCreateSystemThread
                                                  function
        KSTART_ROUTINE sub_140235C44;          // unnamed checker subroutine
        KDPC KiBalanceSetManagerPeriodicDpc;   // global DPC struct
} KI_FILTER_FIBER_PARAM, *PKI_FILTER_FIBER_PARAM;
```

"KiFatalExceptionFilter" stores these pointers to "HalReserved" fields.

3. Creating patchguard context

Let's get back to the "KiFilterFiberContext" function. It's pseudocode is given below:

```
BOOLEAN KiFilterFiberContext(PVOID pKiFilterParam)
{
	BOOLEAN Result = TRUE;
	DWORD64 dwDpcIdx1 = __rdtsc() % 13;
	DWORD64 dwRand2 = __rdtsc() % 10;
	DWORD64 dwMethod1 = __rdtsc() % 6;

	AntiDebug();

	// Let's call sub_1406D6F78 KiInitializePatchGuardContext since it does initialize patchguard context
	Result = KiInitializePatchGuardContext(dwDpcIdx, dwMethod1, (dwRand2 < 6) + 1, pKiFilterParam, TRUE);

	// A 50% chance to create two patchguard contexts
	if (dwRand2 < 6)
	{
		DWORD64 dwDpcIdx2 = __rdtsc() % 13;
		DWORD64 dwMethod2 = __rdtsc() % 6;

		do
		{
			dwMethod2 = __rdtsc() % 6;
		}
		while ((dwMethod1 != 0) && (dwMethod1 == dwMethod2));

		Result = KiInitializePatchGuardContext(dwDpcIdx2, dwMethod2, 2, pKiFilterParam, FALSE);
	}

	AntiDebug();

	return Result;
}
```

It is rather clear, and with provided code we can assume that up to 4 patchguard contexts can be active on a running system simultaneously. Remember this one because wherever it is called, we can be 100% sure that a new patchguard context is being initialized.

The function that creates and initializes patchguard context is so-called "KiInitializePatchGuardContext". It is a huge obfuscated function. I guess it is suitable to reference Alex's Ionescu tweet about it:

"I love the new #Windows 8 Patch Guard. Fixes so many of the obvious holes in downlevel, and the new hyper-inlined obfuscation makes me cry."

You bet it! IDA Pro's decompiler works on it ~20 min on 3770 Core i7 CPU and spews out 26K lines of code. It is not worth dealing with it as a single unit. Luckily, you can bite out small pieces of information that give you a clue about methods that the new patchguard uses. That's why we did not reverse engineer it entirely, as instead we took and analyzed several parts in it. Feel free to explore this function yourself, and you may discover new wonderful things!

It takes 5 parameters on Windows 8.1:

1. Index of DPC routine to be called from a created patchguard DPC for checking the patchguard context. It may be one of these:

    // These ones don't use exception handlers to fire checks

    KiTimerDispatch (copied to random pool allocation)

    KiDpcDispatch (copied into patchguard context)

    // These use exception handlers to fire patchguard checks

    ExpTimerDpcRoutine

    IopTimerDispatch

    IopIrpStackProfilerTimer

    PopThermalZoneDpc

    CmpEnableLazyFlushDpcRoutine

    CmpLazyFlushDpcRoutine

    KiBalanceSetManagerDeferredRoutine

    ExpTimeRefreshDpcRoutine

    ExpTimeZoneDpcRoutine

    ExpCenturyDpcRoutine

Also those 10 DPCs are regular system DPCs with useful payload, but when they encounter a DeferredContext which has non-canonical address, they fire a corresponding KiCustomAccessRoutine function.

These functions are only called when an appropriate scheduling method is used (0, 1, 2, 5)

2. Scheduling method:

These are the methods that are used to fire a patchguard DPC object that is created inside "KiInitializePatchGuardContext" function.

0 - KeSetCoalescableTimer

A timer object is created with a random fire period between 2 minutes and 2 minutes and 10 seconds.

1 - Prcb.AcpiReserved

In this case a patchguard DPC is fired when a certain ACPI event occurs, f.e. transitioning to idle state. In this case "HalpTimerDPCRoutine" checks if 2 minutes have passed since last queued by itself DPC, and queues another one, taken from Prcb.AcpiReserved field.

2 - Prcb.HalReserved

Here a patchguard DPC is queued when HAL timer clock interrupt occurs, in the "HalpMcaQueueDpc". It is also done with 2 minutes period at least. Queued patchguard DPC is taken from Prcb.HalReserved field.

3 - PsCreateSystemThread

In this case, patchguard DPC routine is not used, as instead a system thread is created. The thread procedure is taken from KI_FILTER_FIBER_PARAM structure. Patchguard DPC in turn is used just as a container of the address of a newly created patchguard context.

4 - KeInsertQueueApc

This time a regular kernel APC is queued to the one of the system threads with "KiDispatchCallout" APC procedure. No patchguard DPC is fired also. System thread is chosen based on its start address, i.e. it must be equal to either PopIrpWorkerControl or CcQueueLazyWriteScanThread.

5 - KiBalanceSetManagerPeriodicDpc

Patchguard DPC is stored in a global variable named "KiBalanceSetManagerPeriodicDpc". It is queued in "KiUpdateTimeAssist" function and "KeClockInterruptNotify" function within every "KiBalanceSetManagerPeriod" ticks.

3. This parameter can be either 1 or 2. We are not sure about how it affects "KiInitializePatchGuardContext" function, but it is somehow connected to the quantity of checks being done during patchguard context verification routine execution.

4. A pointer to KI_FILTER_FIBER_PARAM structure. It is noticeable that a method chosen inside "KiInitializePatchGuardContext" is selected based on the presence of this parameter. If it is present, a method bit mask is tested with 0x29 (101001b) which allows methods 0, 3 and 5. Otherwise methods 0, 1, 2 and 4 are available. That makes sense, because methods 3 and 5 require a valid KI_FILTER_FIBER_PARAM structure.

5. Boolean parameter which tells if NT kernel functions checksums have to be recalculated.

As you might guess, the only scheduling method that can be initialized twice is 0, so "KiFilterFiberContext" takes this fact into account when chooses a method for a second call of "KiInitializePatchGuardContext".

4. Firing a patchguard check

4.1. Methods that fire patchguard DPC

The main principle of patchguard check routine is to launch a patchguard context verification routine on a DPC level, and then queue a work item that will check vital system structures on a passive level with a proceeding context recreation and rescheduling. The verification work item uses a copy of "FsRtlUninitializeSmallMcb" function. You can check this one out, if you want to figure out how the check works.

For the methods which use DPC activation there is a common code inside 10 listed DPC routines, which checks "DeferredContext" for being a non-canonical address. If it is OK, DPC just executes its payload. Otherwise one of 10 "KiCustomAccessRoutineX" functions is called. When "KiCustomAccessRoutineX" is called, (last 2 bits + 1) of "DeferredContext" are taken and used to roll along "KiCustomRecurseRoutineX". These recursive routines are cycled incrementing X value. When the roll is over, "KiCustomRecurseRoutineX" tries to dereference a DeferredContext value as a pointer, which inevitably generates #GP exception since this address is non-canonical.

```
// Inside DPC routine
if ( (DeferredContext >> 47) < 0xFFFFFFFFFFFFFFFFui64 && DeferredContext >> 47 != 0 )
// Is DeferredContext a canonical address
{
	...
	KiCustomAccessRoutineX(DeferredContext);
	...
}

void KiCustomAccessRoutine9(DWORD64 DeferredContext)
{
	return KiCustomRecurseRoutine9((DeferredContext & 3) + 1, DeferredContext);
}

void KiCustomRecurseRoutine9(DWORD dwRoll, DWORD64 DeferredContext)
{
	DWORD dwNextRoll;
	DWORD64 go_go_GP;

	dwNextRoll = dwRoll - 1;
	if ( dwNextRoll )
		KiCustomRecurseRoutine0(dwNextRoll, DeferredContext);
```

```
	go_go_GP = *DeferredContext; // #GP
}

// DPC routine call sequence
ExpTimerDpcRoutine -> KiCustomAccessRoutine0 -> KiCustomRecurseRoutine0 ... KiCustomRecurseRoutineN
IopTimerDispatch -> KiCustomAccessRoutine1 -> KiCustomRecurseRoutine1 ... KiCustomRecurseRoutineN
IopIrpStackProfilerTimer -> KiCustomAccessRoutine2 -> KiCustomRecurseRoutine2 ... KiCustomRecurseRoutineN
PopThermalZoneDpc -> KiCustomAccessRoutine3 -> KiCustomRecurseRoutine3 ... KiCustomRecurseRoutineN
CmpEnableLazyFlushDpcRoutine -> KiCustomAccessRoutine4 -> KiCustomRecurseRoutine4 ... KiCustomRecurseRoutineN
CmpLazyFlushDpcRoutine -> KiCustomAccessRoutine5 -> KiCustomRecurseRoutine5 ... KiCustomRecurseRoutineN
KiBalanceSetManagerDeferredRoutine -> KiCustomAccessRoutine6 -> KiCustomRecurseRoutine6 ... KiCustomRecurseRoutineN
ExpTimeRefreshDpcRoutine -> KiCustomAccessRoutine7 -> KiCustomRecurseRoutine7 ... KiCustomRecurseRoutineN
ExpTimeZoneDpcRoutine -> KiCustomAccessRoutine8 -> KiCustomRecurseRoutine8 ... KiCustomRecurseRoutineN
ExpCenturyDpcRoutine -> KiCustomAccessRoutine9 -> KiCustomRecurseRoutine9 ... KiCustomRecurseRoutineN
```

Here comes vectored exception handling again. If you look up all the exception handlers for these DPC routines, you'll discover that there are several nested __try__except and __try__finally blocks. For example, "ExpTimerDpcRoutine" looks something like this:

```
...
__try
{
	__try
	{
		__try
		{
			__try
			{
				KiCustomAccessRoutine0(DeferredContext);
			}
			__finally
			{
```

```
                        FinalSub1();
                }
        }
        __except (FilterSub1())    // patchguard context decryption occurs here
        {
                // Nothing
        }
    }
    __finally
    {
        FinalSub2();
    }
}
__except (FilterSub2())
{
    // Nothing
}
...
```

ExpCenturyDpcRoutine, ExpTimeZoneDpcRoutine, ExpTimeRefreshDpcRoutine, KiBalanceSetManagerDeferredRoutine, CmpLazyFlushDpcRoutine, CmpEnableLazyFlushDpcRoutine, PopThermalZoneDpc, ExpTimerDpcRoutine … -/-> _C_specific_handler

IopIrpStackProfilerTimer , IopTimerDispatch … -/-> _GSHandlerCheck_SEH (GS check + _C_specific_handler)

Depending on the DPC routine, decryption routine (based on KiWaitAlways and KiWaitNever variables) may reside in one of the exception filters, exception handlers or termination handlers. Further patchguard context verification occurs also inside decryption routine, right after the decryption.

As for "KiTimerDispatch" and "KiDpcDispatch" DPC routines - they call patchguard context verification directly. Also, depending on the DPC routine a different type of patchguard context encryption is used (or not used at all).

4.2. Other methods

Method 3 creates a system thread. System thread procedure sleeps between 2 minutes and 2 minutes and 10 seconds using "KeDelayExecutionThread" or "KeWaitForSingleObject" on a kernel object, which is always not signaled. After the wait is timed out it decrypts patchguard context and executes verification routine.

Method 4 inserts an APC with "KiDispatchCallout" function as a kernel routine and "EmpCheckErrataList" as a normal routine. Patchguard context decryption and validation occurs

upon APC delivery to the target waiting thread, which happens almost immediately. A 2 minutes wait is located inside the verifier work item routine in this method.

5. One more piece of a puzzle

That would be it about patchguard initialization, but looking for the cross-references to KUSER_SHARED_DATA.KdDebuggerEnabled lead me to a suspicious function named "CcInitializeBcbProfiler". It is full of bit rotations and magic numbers, which forced me to check whether it is related to patchguard mechanism.

... --> Phase1InitializationDiscard --> CcInitializeCacheManager --> CcInitializeBcbProfiler

It seems to have the same roots!

With 50% chance it queues DPC with "CcBcbProfiler" routine or a work item with an unnamed work item routine (which is almost identical to the "CcBcbProfiler" routine). This mechanism picks one random function from NT kernel module and checks its consistency every 2 minutes.

It is interesting that all of the patchguard-related functions are located nearby, one after another starting from "FsRtlMdlReadCompleteDevEx". It tells us that they are likely to be located in a single compilation unit. This fact gives us a hope that all of the patchguard initialization paths have been covered in this article.

6. Attacks

Now, as we covered patchguard initialization, we know what wires of a patchguard bomb can be cut to defuse it! However, there are several ways depending on a patchguard DPC scheduling method. Since we cover a specific version of patchguard, i.e. Windows 8.1, we are going to use precomputed offsets for accessing the private kernel structures' fields.

The common defusing principle is firstly to check if verification routine is in progress, and wait a bit if it is true. Then do the following:

0 - KeSetCoalescableTimer. Scan through the Prcb timer table and disable the one with suitable DPC object.

1 - AcpiReserved field. Zero this field out, so the DPC won't be fired again.

2 - HalReserved field. Same here.

3 - PspCreateSystemThread. Enumerate all threads in a system and unwind their stacks. Then check if a start routine from “KiServiceTablesLocked” structure is present in a call stack. If it is there, it's a patchguard thread. Disable it while it is in a wait state setting the wait time to infinite.

4 - APC. Take the current Prcb NUMA Node and its worker thread pool. Scan through its sleeping worker threads unwinding the stacks until "ExpWorkerThread" function. If there are functions that are not to be found in NT image runtime function data, try to unwind them sequentially with runtime data for "FsRtlMdlReadCompleteDevEx" and "FsRtlUninitializeSmallMcb". If succeeded, than it is a patchguard worker. Disable it setting the wait time to infinity.

5 - KiBalanceSetManagerPeriodicDpc. Zero this struct out.

By disabling a timer we mean setting its due time to infinity, so it never fires. And by suitable DPC object we mean a DPC object with a deferred context set to a non-canonical address. Furthermore, you can additionally check this pointer to be valid after XORing its value with a quad-word following right after KDPC struct and ANDing it with 0xFFFF800000000000.

As for the "CcBcbProfiler" piece, we consider it not to be relevant since there is a small chance that it will check exactly the needed function.

7. Summary

A quality of Windows 8.1 kernel patch protection mechanism is extremely high. There are a lot of interesting anti-debugging tricks used again dynamic analysis, f.e. resetting IDT before accessing debug registers (which leads you to hanging if you set break on debug registers access), overall obfuscation like using macroses for generating pseudo-random values, loop unrolling etc. It is also extremely difficult to do a static analysis since a lot of indirect function calls are used including the usage of exception handlers.

It is a really nice tool to keep the system safe. Therefore we hope that as a developer you won't face situations when you need to disable this cool mechanism!

References

1. Andrea Allievi. "The Windows 8.1 Kernel Patch Protection". VRT blog. August 14, 2014.

2. Skape, Bypassing PatchGuard on Windows x64, Uninformed, December 2005

3. Skywing, PatchGuard Reloaded - A Brief Analysis of PatchGuard Version 3, Uninformed, September 2007

4. Christoph Husse, Bypassing PatchGuard 3 - CodeProject, August 2008

About Positive Technologies and Research Center

1.1. About Positive Technologies

Positive Technologies is a leading provider of vulnerability assessment, compliance management and threat analysis solutions to more than 1,000 global enterprise clients. Our solutions work seamlessly across your entire business: securing applications in development; assessing your network and application vulnerabilities; assuring compliance with regulatory requirements; and blocking real-time attacks. Our commitment to clients and research has earned Positive Technologies a reputation as one of the foremost authorities on SCADA, Banking, Telecom, Web Application and ERP security, and distinction as the #1 fastest growing Security and Vulnerability Management firm in 2012, as shown in an IDC report.

To learn more about Positive Technologies please visit: http://www.ptsecurity.com

1.2. About Positive Research Center

Positive Research, is one of the largest and most dynamic security research facilities in Europe. Our award-winning center carries out research, design and analytical work and threat and vulnerability analysis. Our compliance and vulnerability management solutions are updated to provide up-to-the-minute protection as new vulnerabilities are detected.

Our expert researchers are the advanced white hats in the industry, detecting more than one hundred 0 day vulnerabilities per year in leading products such as operating systems, network equipment and applications.

We help leading technology companies like Apple, Cisco, Google, Microsoft, Oracle, SAP and VMware eliminate vulnerabilities from their software that threatens the security of your systems.

http://www.ptsecurity.com/research/

Our Vulnerability Alerts:

http://www.ptsecurity.com/research/advisory/